Feargus Pendlebury

Machine Learning for Security in Hostile Environments

Royal Holloway, University of London
Thesis submitted for the degree of Doctor of Philosophy

Declaration of Authorship

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Information Security Group as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award at any other university or educational establishment.

Feargus Pendlebury
September, 2021

Publications

The content of this thesis includes ideas and findings previously presented in the following publications. An asterisk (*) denotes joint lead authorship.

1. Pendlebury F.*, Pierazzi F.*, Jordaney R., Kinder J., Cavallaro L. Enabling Fair ML Evaluations for Security. In *Proc. of the ACM Conference on Computer and Communications Security (CCS) (poster)*. 2018.

2. Pendlebury F.*, Pierazzi F.*, Jordaney R., Kinder J., Cavallaro L. TESSERACT: Eliminating Experimental Bias in Malware Classification Across Space and Time. In *Proc. of the USENIX Security Symposium*. 2019.

3. Pierazzi F.*, Pendlebury F.*, Cortellazzi J., Cavallaro L. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proc. of the IEEE Symposium of Security and Privacy (S&P)*. 2020.

4. Labaca-Castro R., Muñoz-Gonzàlez L., Pendlebury F., Rodosek G. D., Pierazzi F., Cavallaro L. Universal Adversarial Perturbations for Malware. In *arXiv CoRR repo. (preprint)*. 2021.

5. Arp, D., Quiring E., Pendlebury F., Warnecke A., Pierazzi F., Wressnegger C., Cavallaro L., Rieck K. Dos and Don't of Machine Learning in Computer Security. To appear in *Proc. of the USENIX Security Symposium*. 2022.

6. Barbero F.*, Pendlebury F.*, Pierazzi F., Cavallaro L. Transcending TRANSCEND: Revisiting Malware Classification in the Presence of Concept Drift. To appear in *Proc. of the IEEE Symposium of Security and Privacy (S&P)*. 2022.

Additionally, the author has contributed to the following publications which further discuss some of the concepts presented in this thesis.

7. Andresini G., Pendlebury F., Pierazzi F., Loglisci C., Appice A., Cavallaro L. INSOMNIA: Towards Concept-Drift Robustness in Network Intrusion Detection. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. 2021.

8. Kan Z., Pendlebury F., Pierazzi F., Cavallaro L. Investigating Labelless Drift Adaptation for Malware Detection. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. 2021.

Research for publications ##1–6 and ##7–8 were partly carried out while a visiting scholar at King's College London and University College London, respectively. Research for publication #4 was partly carried out while a research visitor at the Alan Turing Institute and research for publications ##7–8 was partly carried out while a research intern at the International Computer Science Institute, Berkeley.

All work has been supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP P009301/1).

Acknowledgements

"Alone we can do so little.
Together we can do so much."
—Helen Keller

Little can be achieved from working in isolation and it is my fortune to have been supported by many great researchers, engineers, and friends. While there are bound to be some omissions below, I will do my best to try and give props where they are due.

Firstly I must thank my supervisor Lorenzo Cavallaro. In his thesis he humbly wrote: *"I am far from being a good researcher, but I will do all of my best to become it"*. I can only confirm that he has achieved this many times over and that it can only be my goal to become one iota the talented, hard-working, and kind-spirited researcher that he is.

Additionally I am beholden to Johannes Kinder who supervised me alongside Lorenzo before moving to a new home in Munich, and who bestowed upon me much invaluable wisdom. He remains the coolest and most unflappable researcher I've met yet.

I am indebted to the heavyweight tag team titleweight holders: Kenny Paterson for supervising the first year of my PhD and Martin Albrecht for supervising the final stretch, and to both of them for remaining sources of wisdom, inspiration, and guidance throughout my studies.

I am lucky to have had the chance to work alongside Fabio Pierazzi, a great friend who mentored me on the science and art of paper writing, and greatly accelerated my growth as a researcher.

I am grateful to Emil Tan who took me further down the rabbit hole of security and encouraged me to pursue a PhD on the topic, and to the oncology patient at Mount Alvernia who first introduced me to computer science years before—without these acts of serendipity, I would not be here today.

Warm thanks especially go to Suman Jana, Jorge Blasco Alis, and Carlos Cid for agreeing to act as my PhD committee and for reading my work with scrutiny.

I owe much to the various visits and collaborations throughout the PhD. For the endless laughs, insightful conversations, and comforting exchange of radical thought, I thank friends from the Alan Turing Institute: George Elder, Sam Van Stroud, Victor Darvariu, Alicia Cork, Markus Löning, and Tiejun Wei.

I thank David Freeman from Facebook for giving me the opportunity to apply machine learning to real world security issues and to Daniel Bernhardt, Alexander Erofeev, Mikhail Pershin, Michael

Blind, Sharon Zheng, Christoph Besel, Estee van der Walt, Mark Atherton, Nedyalko Prisadnikov, Maria Kirichun, Despoina Magka, Cihad Öge, Alexis Woo, Michał Staszewski, Isaac Kamlish, Vikram Padval, Alex Vaystikh, and everyone else that I had the great fortune of learning from while interning there.

I am very grateful to Sadia Afroz and Michael Mahoney at ICSI and UC Berkeley for welcoming me into their group, as well as Francisco Utrera and Yiğitcan Kaya for the great brainstorming sessions while working on the TrojAI competition.

Additionally I thank King's College London and University College London for hosting me as a visiting scholar and allowing me to follow Lorenzo on his journey as a professor.

I would also like to acknowledge my coauthors and collaborators for their tenacity and insight during our work together: Sadia Afroz, Giuseppina Andresini, Annalisa Appice, Daniel Arp, Yavuz Bakman, Federico Barbero, Brano Bosansky, Lorenzo Cavallaro, Zhi Chen, Jacopo Cortellazzi, Ben Erichson, Roberto Jordaney, Zeliang Kan, Yiğitcan Kaya, Johannes Kinder, Raphael Labaca-Castro, Corrado Loglisci, Samaneh Mahdavifar, Michael Mahoney, Luis Muñoz-González, Fabio Pierazzi, Erwin Quiring, Harivallabha Rangarajan, Gabi Dreo Rodosek, Petr Somol, Flavio Toffalini, Liang Tong, Francisco Utrera, Yevgeniy Vorobeychik, Gang Wang, Alexander Warnecke, Christian Wressnegger, Da Yang, Limin Yang, Yaoqing Yang, and Ziqi Yang.

I am most fortunate to have embarked on the PhD journey with a CDT cohort, who have irreversibly enriched my life: Fernando Virdia, Rob Markiewicz, Eamonn Postlethwaite, Rory Hopcraft, Pallavi Sivakumaran, Ashley Fraser, and Pete Beaumont.

I also thank my labmates and CTF teammates for their great company and for their aid in scheming about a future farm life: Simon, Roberto, Claudio, Gio, Dusan, James, Jason, Blake, Duncan, Guillermo, Santanu, Emanuele, Salah, and other members of the oldskool, plus of course Federico, Jacopo, Mark, Giulio, and members of the new.

I am grateful to my mum Tessa and sister Gemma for their consistent support and belief throughout the years and to my friends from outside academia (*i.e.*, the real world), particularly Tawqir, Kosi, Vasu, Neil, and Tom,[1] for the many joyful memories and for reminding me of what's important in life.

[1] with special mention to Pebble and Calvin

Finally I especially thank my partner Steph, whose patience and loving encouragement form the structural support on which the following pages sit—the end of this thesis marks the start of many more adventures together.

Dedicated to the memory of

Don Pendlebury
and
Kai Luotsinen

Abstract

The potential for machine learning to change the world is undeniable. More data, better resources, and advances in algorithms have led to multiple breakthroughs in fields such as computer vision and natural language processing. Recently, efforts have been made to apply these methods to detection tasks in computer security where a system detects the presence of malicious objects to prevent them from causing harm to users. However the problem is challenging, primarily due to the *inherently hostile environments* that security detectors are deployed to. In these settings, members of the malicious class actively try to avoid detection, leading to drift in the data distribution over time that violates core assumptions of machine learning. Furthermore, adversaries can apply powerful algorithms to search for *adversarial examples*: objects which are confidently misclassified as benign by a detector while retaining malicious functionality.

In this thesis we explore whether machine learning is ready to be used in the security domain, given this hostile environment. We outline how adversarial behavior manifests in security data, providing novel perspectives on the relationship between concept drift and adversarial examples, as well as between feature-space and problem-space adversarial attacks. These lead us to devise a new problem-space attack demonstrating that adversarial examples are a realistic and practical threat against malware detectors.

We discuss the difficulty in performing fair, informative evaluations of defenses in such a dynamic and volatile environment, showing how the evaluations of previous state-of-the-art detectors have been inflated by experimental bias. Through an examination of these issues we construct actionable guidance on how to alleviate bias, allowing for clearer comparisons between drift mitigations. Finally, we propose a framework for classification with rejection based on conformal prediction and conformal evaluation theory which is able to identify and quarantine drifting examples, improving on previous work in terms of performance and runtime cost.

Ultimately we find that the benefits of machine learning remain a tantalizing solution for security detection. While challenges remain, the application of mechanisms to identify, track, and adapt to drifting and adversarial inputs—if realistic evaluations are used to assess them—can greatly raise the bar for attackers.

Contents

List of Figures

List of Tables

List of Symbols

Symbols from Chapter 4

Table 1 is a legend of the main symbols used throughout Chapter 4.

Symbol	Description
$\mathcal{Z}$	Problem space (i.e., input space).
$\mathcal{X}$	Feature space $\mathcal{X} \subseteq \mathbb{R}^n$.
$\mathcal{Y}$	Label space.
φ	Feature mapping function $\varphi : \mathcal{Z} \longrightarrow \mathcal{X}$.
h_i	Discriminant function $h_i : \mathcal{X} \longrightarrow \mathbb{R}$ that assigns object $\boldsymbol{x} \in \mathcal{X}$ a score in $\mathbb{R}$ (e.g., distance from hyperplane) that represents fitness to class $i \in \mathcal{Y}$.
g	Classifier $g : \mathcal{X} \longrightarrow \mathcal{Y}$ that assigns object $\boldsymbol{x} \in \mathcal{X}$ to class $y \in \mathcal{Y}$. Also known as *decision function*. It is defined based on the output of the discriminant functions $h_i, \forall i \in \mathcal{Y}$.
$\mathcal{L}_y$	Loss function $\mathcal{L}_y : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ of object $\boldsymbol{x} \in \mathcal{X}$ with respect to class $y \in \mathcal{Y}$.
$f_{y,\kappa}$	Attack objective function $f_{y,\kappa} : \mathcal{X} \times \mathcal{Y} \times \mathbb{R} \longrightarrow \mathbb{R}$ of object $\boldsymbol{x} \in \mathcal{X}$ with respect to class $y \in \mathcal{Y}$ with maximum confidence $\kappa \in \mathbb{R}$.
f_y	Compact notation for $f_{y,0}$.
Ω	Feature-space constraints.
$\boldsymbol{\delta}$	$\boldsymbol{\delta} \in \mathbb{R}^n$ is a symbol used to denote a feature-space perturbation vector.
$\boldsymbol{\eta}$	Side-effect feature vector.
T	Transformation $T : \mathcal{Z} \longrightarrow \mathcal{Z}$.
$\mathbf{T}$	Transformation sequence $\mathbf{T} = T_n \circ T_{n-1} \circ \cdots \circ T_1$.
$\mathcal{T}$	Space of available transformations.
Y	Suite of automated tests $\tau \in \mathrm{Y}$ to verify preserved semantics.
Π	Suite of manual tests $\pi \in \Pi$ to verify plausibility. In particular, $\pi(z) = 1$ if $z \in \mathcal{Z}$ is plausible, else $\pi(z) = 0$.
Λ	Set of preprocessing operators $\mathbf{A} \in \Lambda$ for which $z \in \mathcal{Z}$ should be resistant (i.e., $\mathbf{A}(\mathbf{T}(z)) = \mathbf{T}(z)$).
Γ	Problem-space constraints Γ, consisting of $\{\Pi, \mathrm{Y}, \mathcal{T}, \Lambda\}$.
$\mathcal{D}$	Training dataset.
$\boldsymbol{w}$	Model hyper-parameters.
Θ	Knowledge space.
θ	Threat model assumptions $\theta \in \Theta$; more specifically, $\theta = (\mathcal{D}, \mathcal{X}, g, \boldsymbol{w})$. A *hat* symbol is used if only estimates of parameters are known. See Section 2.3 for more details.

Table 1: Chapter 4 symbols.

Symbols from Chapter 5

Table 2 is a legend of the main symbols used throughout Chapter 5.

Symbol	Description
gw	Short version of goodware.
mw	Short version of malware.
ML	Short version of Machine Learning.
D	Labeled dataset with malware (mw) and goodware (gw).
Tr	Training dataset.
W	Size of the time window of the training set (*e.g.*, 1 year).
Ts	Testing dataset.
S	Size of the time window of the testing set (*e.g.*, 2 years).
Δ	Size of the test time-slots for time-aware evaluations (*e.g.*, months).
AUT(f,N)	Area Under Time, a new metric we define to measure performance over time decay and compare different solutions (subsection 5.5.2). It is always computed with respect to a performance function f (*e.g.*, F_1-Score) and N is the number of time units considered (*e.g.*, 24 months)
$\hat{\sigma}$	Estimated percentage of malware (mw) in the wild.
φ	Percentage of malware (mw) in the training set.
δ	Percentage of malware (mw) in the testing set.
$\mathbb{P}$	Performance target of the tuning algorithm in subsection 5.5.3; it can be F_1-Score, Precision (*Pr*) or Recall (*Rec*).
$\varphi^*_{\mathbb{P}}$	Percentage of malware (mw) in the training set, to improve performance $\mathbb{P}$ on the malware (mw) class (subsection 5.5.3).
E	Error rate (subsection 5.5.3).
E_{max}	Maximum error rate when searching $\varphi^*_{\mathbb{P}}$ (subsection 5.5.3).
Θ	Model learned after training a classifier.
L	Labeling cost.
Q	Quarantine cost.
P	Performance; depending on the context, it will refer to AUT with F_1 or *Pr* or *Rec*.

Table 2: Chapter 5 symbols.

Symbols from Chapter 6

Table 3 is a legend of the main symbols used throughout Chapter 6.

Symbol	Description
$\mathcal{X}$	Feature space $\mathcal{X} \subseteq \mathbb{R}^n$.
$\mathcal{Y}$	Label space.
z	Example pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$.
z^*	Previously unseen testing example.
$\hat{y}$	Predicted class $g(z^*)$.
a_z	Nonconformity score output by an NCM for z.
p_z	Statistical p-value for z.
p_z^y	Statistical p-value for z, calculated with respect to class $y \in \mathcal{Y}$ (used in *label conditional* calculations).
τ_y	A rejection threshold $\tau_y \in [0, 1]$ for class $y \in \mathcal{Y}$.
$\mathcal{T}$	The set of all per-class rejection thresholds $\{ \tau_y \in [0, 1] \mid y \in \mathcal{Y} \}$.
B	Bag of examples $\wr z_1, z_2, ..., z_n \wr$.
d	Distance function $d(z, z')$.
$\hat{z}$	Point predictor $\hat{z}(B)$.
A	Nonconformity measure (NCM) usually composed of a distance function and point predictor.
S	Collection of nonconformity scores computed in elements of B, relative to other elements in B, $S = \wr A(B \setminus \wr z \wr, z) : z \in B \wr$.
g	Classifier $g : \mathcal{X} \longrightarrow \mathcal{Y}$ that assigns object $x \in \mathcal{X}$ to class $y \in \mathcal{Y}$. Also known as the *decision function*.
ε	Significance level used in conformal prediction to define prediction region with confidence guarantees.
NCM	Nonconformity measure.
TCE	Transductive Conformal Evaluator.
ICE	Inductive Conformal Evaluator.
CCE	Cross-Conformal Evaluator.

Table 3: Chapter 6 symbols.

Code Availability

Adversarial Program Generation

Chapter 4 proposes a novel Android evasion attack based on automated software transplantation. We release the code and data of our approach to other researchers by responsibly sharing a private repository. The project website with instructions to request access is at: `https://s2lab.cs.ucl.ac.uk/projects/intriguing`.

Tesseract Evaluation Framework

Chapter 5 discusses Tesseract, our framework for conducting time-aware evaluations free from spatio-temporal bias. We make Tesseract's code and data available to the research community to promote the adoption of a sound and unbiased evaluation of classifiers. The Tesseract project website with instructions to request access is at: `https://s2lab.cs.ucl.ac.uk/projects/tesseract`.

Transcendent Drift Identification

Chapter 6 introduces Transcendent, an extension of the Transcend framework [139] for identifying and rejecting drifting examples. To assist researchers and practitioners alike, we release the code of Transcendent as open source, along with the data used in the empirical evaluations, to aid with the scalable identification and remediation of concept drift: `https://s2lab.cs.ucl.ac.uk/projects/transcend`.

A note on impact. To date, code from these projects has been used by groups from across various academic and industrial institutions, including: the Alan Turing Institute, Beijing University of Posts and Telecommunications, Birla Institute of Technology and Science, Boise State University, Capital One, Carnegie Mellon University, Columbia University, Czech Technical University, Deakin University, the Federal University of Paraná, Fudan University, Georgia Tech, Guangzhou University, the Indian Institute of Information

Technology and Management, the Institute for Information Industry, the International Computer Science Institute, the Hong Kong Polytechnic University, the Hong Kong University of Science and Technology Library, Huazhong University of Science and Technology, Karlsruhe Institute of Technology, King's College London, Korea University, the MITRE Corporation, Nanjing University, National Cybersecurity Agency of France, the National Institute of Technology, National University of Defense Technology, New York University, Northeastern University, Northwest University, Orange Labs, Osaka University, PSG College of Technology, Queen's University Belfast, Reichman University , Rice University, Royal Holloway University of London, the Swinburne University of Technology, Tsinghua University, TU Dublin, TU Berlin, TU Braunschweig, TU Munich, University of Adelaide, University of Bari Aldo Moro, University of British Columbia, University of Cagliari, University of California Berkeley, University of California Santa Barbara, University of Cambridge , University Carlos III of Madrid , University College London, University of the Fraser Valley, University of Illinois at Urbana-Champaign, University of Jinan, University of Luxembourg, University of Michigan, University of New South Wales, University of Oregon, University of Rennes 1, University of Toronto, University of Virginia, University of Wisconsin-Madison, VIT Bhopal, Washington State University, Washington University in St. Louis, Wuhan University, Xidian University, Zhejiang Gongshang University, and Zhejiang University.

Part I:

Prologue

> [...] you don't understand things. You just get used to them.
>
> John von Neumann

1 Overview

Machine learning is certainly changing the world. Wider access to computational resources, enormous repositories of data, and novel concepts and system architectures have paved the way for multiple breakthroughs in various research fields such as computer vision [156, 127, 261] and natural language processing [278, 24, 298, 79]. These advances are leading to new technology previously confined to science fiction, including self-driving vehicles [41], smart lenses [151, 180], and universal translators [285, 18].

Machine learning has clear power and potential...

The effect of this technology on society may be profound. By playing a part in reducing marginal costs and the need for labor exploitation through scalability and automation, learning-based technologies could facilitate a transition to postcapitalist society [189, 25]. At the other extreme, machine learning has also opened the doors to new methods of surveillance [117, 95], censorship [262, 205, 29, 238], and disinformation [3, 60].

Given the clear power and potential of machine learning, it seems natural to apply the same methodologies to solve detection tasks in information security. In detection tasks, a set of malicious objects need to be identified and separated from a collection of benign objects to prevent them from causing harm to users (e.g., predicting whether a new email is spam or legitimate). The detection task has many similarities to well-studied tasks from the aforementioned fields, such as image classification in computer vision or named entity recognition in natural language processing, so the problem itself seems well-suited for machine learning.

...which motivates its application to security detection tasks.

As well as performance gains in accuracy, machine learning has the potential to automate expensive manual processes, such as reverse engineering of individual malware, freeing up resources and allowing for more flexibility in responding to threats. The power to generalize to new and unforeseen threats makes machine learning-based approaches more resilient than previous rule-based methods, and the ability to extract and explain patterns in data can help analysts better understand the threat environment [305] and can even be used to support older solutions by generating highly-precise, yet brittle, signatures [233].

Indeed there has been progress, with numerous success stories across various research areas such as network intrusion detection [83, 196, 13], code authorship attribution [2, 136, 44], and web

security [274, 134, 149]. A particularly active area as been malware detection, for which solutions have been proposed to detect malware for Android [187, 19, 7, 195], Windows PE [11, 232, 281], Javascript [70, 153, 92], and PDF [271, 272, 54, 265] domains.

Such solutions are not just academic, with the security teams of large technology corporations deploying learning-based solutions to protect their services. For example, Facebook use machine learning to detect abusive accounts which are used spread spam, malware, and disinformation [317, 125] while Google [116], Apple [15], and Microsoft [192] all use machine learning-based malware detectors to prevent malicious apps gaining traction on their platforms.

However, security detectors face hostile environments where adversaries seek to evade detection...

However, this progress has not been without challenges, primarily due to the *hostile environments* that security detectors are deployed to. Adversaries that seek to manipulate and deceive learning-based systems certainly exist for traditional tasks, for example, an attacker may add a specially-crafted sticker to a traffic sign to cause the recognition model of a self-driving vehicle to misclassify it as a different sign [89]. However, in security detection tasks the environment is *inherently* hostile, that is, the positive class which is the target of the detection *has agency* and *does not want to be detected*. Consider again the traffic sign example. All other things being equal, if an update is pushed to the recognition model to improve it in some way, the distribution of traffic signs does not change—a stop sign is still a stop sign and a one-way sign still a one-way sign. The signs themselves have no agency—signs for crossroads don't try to look like signs for roundabouts, or vice-versa. Unfortunately, the same cannot be said for security detection tasks. For these, the distribution of the positive class responds to any change in the detection model. These changes can be sudden and abrupt, or appear as a gradual evolution over time. Unlike the traffic signs, the definition of the malicious class changes: what was once considered 'definitely malicious' may now look 'possibly malicious' or even 'certainly benign'. In order to ensure their operations remain effective and profitable, malicious actors utilize new obfuscation techniques to evade malware detectors, alter packet statistics to evade network intrusion detectors, and mimic legitimate social media posts to evade spam detection.

...causing drift in the data distribution over time, violating i.i.d.

This evolution of the malicious class over time is a driving factor in the presence of *dataset shift*, or *concept drift*, in security data, where the data distribution diverges from the distribution it was originally modeled as. Although this has been a hallmark of security data long before artificial intelligence was applied to security tasks, it is particularly problematic for systems using machine learning algorithms as these algorithms typically rely on the assumption that new data and the data used for training the model are identically and independently drawn from the same joint distribution (i.i.d.). As this assumption weakens over time, the model performance begins to deteriorate.

There are further challenges still. As hinted at earlier, while the traffic sign recognition setting is not itself inherently hostile, malicious actors can introduce ways to fool the classification model and disrupt the data distribution. This is possible because machine learning algorithms suffer from a suite of limitations centered around the fact that the way they model the classification problem does not perfectly align with our human intuition [132]. That is, there exist examples of say, stop signs, which appear like stop signs to a human but are confidently classified as a one-way sign by the recognition model. Furthermore, an adversary can purposely craft perturbations which when applied to correctly classified examples, will transform them into these misclassified variants—one such attack uses a specially crafted sticker, as mentioned previously. When crafted in this way, these problematic examples are called *adversarial examples* (note that the detection of these perturbations is itself a security detection task).

Additionally, recent strong attacks against ML systems using adversarial examples...

Of course, these attacks are also possible against security detectors that use machine learning. There exist malware which looks like malware to a human, but is misclassified as goodware by the model, and powerful attacks exist to help adversaries find and abuse such examples. While advantageous in so many ways, the use of machine learning for detection has also opened up a whole new attack surface for attackers and as these vulnerabilities are abused and expanded on, the severity and speed of concept drift is further exacerbated.

...also affect security detectors, exacerbating drift.

This thesis explores whether *machine learning is ready to be used in the security domain*, given the hostile environment. It outlines how concept drift and adversarial examples manifest in security data and the performance degradation they can induce in machine learning-based detectors; it discusses the difficulty of performing fair, informative experimentation in such a dynamic and volatile environment; and it proposes mitigations that can be deployed to identify and adapt to the challenging drifting conditions.

1.1 Thesis Organization

The thesis is structured in three parts, first introducing the necessary background (part I), then characterizing the hostile environment faced by learning-based security detectors (part II), and finally exploring the design and evaluation of defenses given this environment (part III). Here we give a brief roadmap through the individual chapters.

This thesis is in three parts...

1.1.1 *Part I: Prologue*

In the first part we define the preliminaries. We begin here, in **Chapter 1**, outlining the problem and providing some orientation for the chapters to come and the contributions therein.

...the first reviews the fundamentals...

Chapter 2 introduces the fundamental concepts of machine learning and security which will help the reader interpret findings from the later chapters. It includes some context on how machine learning algorithms are used for security, and on the security of machine learning algorithms.

1.1.2 *Part II: Adversarial Interactions*

The second part provides a characterization of the 'hostile environment' and focuses on understanding the primary consequences of adversarial interactions with machine learning-based security systems: concept drift and adversarial examples.

Chapter 3 includes an overview of concept drift in security and provides a new perspective on the relationship between drift and adversarial examples. While concept drift is often treated as a 'natural' phenomenon, it is evident that concept drift is driven by adversarial behavior [27] and that there is an intrinsic relationship between the two. In this chapter we present a theoretical framework to unify these different phenomena, which motivates research for their joint mitigation.

Chapter 4 discusses how adversarial examples can be crafted in security detection settings where the generation of adversarial examples is far more constrained than in traditional domains such as computer vision. We propose a novel formalization for describing realizable evasion attacks in the problem-space, and define a comprehensive set of constraints for their generation: available transformations, preserved semantics, robustness to preprocessing, and plausibility. We demonstrate how the formalization facilitates the design of stronger attacks, and propose a novel problem-space attack on Android malware that overcomes past limitations. Findings from this chapter have been previously published in:

...the second characterizes concept drift and adversarial examples in security...

- Pierazzi F.*, Pendlebury F.*, Cortellazzi J., Cavallaro L. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proc. of the IEEE Symposium of Security and Privacy (S&P)*. 2020.

- Labaca-Castro R., Muñoz-Gonzàlez L., Pendlebury F., Rodosek G. D., Pierazzi F., Cavallaro L. Universal Adversarial Perturbations for Malware. In *arXiv CoRR repository (preprint)*. 2021.

1.1.3 *Part III: Detection in a Hostile Environment*

In the final part we reason about how to proceed given the hostile environment that learning-based detectors face. We establish how to design fair, realistic evaluations of defenses and then discuss different methods for mitigating concept drift in malware data distributions.

...and the third provides guidance on ensuring the fair and realistic evaluation of defenses...

Chapter 5 discusses sources of experimental bias that can affect the evaluation of machine learning-based detectors, many of which are caused by idiosyncrasies of the security setting. To begin with we identify *spatio-temporal biases* that have afflicted evaluations of many state-of-the-art Android malware detectors and present Tesseract, a framework for ensuring fair evaluations free from these biases. We explore different mitigations for concept drift such as incremental retraining, active learning, and classification with rejection, and show how Tesseract can be used to compare cost and performance across these different methods. Finally we look beyond Android malware and assess the prevalence of these biases in past work on security detection. Then we broaden the discussion further, presenting ten pitfalls which have frequently not been controlled for in prior work and recommendations for removing them. Findings from this chapter have been previously published in:

- Pendlebury F.*, Pierazzi F.*, Jordaney R., Kinder J., Cavallaro L. Enabling Fair ML Evaluations for Security. In *Proc. of the ACM Conference on Computer and Communications Security (CCS) (poster)*. 2018.
- Pendlebury F.*, Pierazzi F.*, Jordaney R., Kinder J., Cavallaro L. TESSERACT: Eliminating Experimental Bias in Malware Classification Across Space and Time. In *Proc. of the USENIX Security Symposium*. 2019.
- Arp, D., Quiring E., Pendlebury F., Warnecke A., Pierazzi F., Wressnegger C., Cavallaro L., Rieck K. Dos and Don't of Machine Learning in Computer Security. To appear in *Proc. of the USENIX Security Symposium*. 2022.

...as well as exploring promising drift mitigation strategies.

Chapter 6 presents Transcendent, a framework for classification with rejection based on Transcend [139], a rejection strategy based on conformal evaluation theory—an extension of conformal prediction theory adapted for drifting settings. We provide a formal treatment of rejection using conformal evaluation, gaining a better understanding of the theoretical reasons behind its success. We propose two new conformal evaluators—the components that provide statistical support for the rejections—which match or surpass the performance of the original while significantly decreasing the computational overhead. These improvements make Transcend [139] a sound and practical solution for drift mitigation for the first time,

and we demonstrate its efficacy for detecting drifting examples for Android, Windows PE, and PDF malware. Findings from this chapter have been previously presented in:

- Barbero F.*, Pendlebury F.*, Pierazzi F., Cavallaro L. Transcending TRANSCEND: Revisiting Malware Classification in the Presence of Concept Drift. To appear in *Proc. of the IEEE Symposium of Security and Privacy (S&P)*. 2022.

2 *Machine Learning and Security*

There are two sides to machine learning and security. In the first, machine learning can be applied to security tasks, such as for malware detection or network intrusion detection—this is machine learning for security. In the second, machine learning algorithms themselves have vulnerabilities which can be exploited, require security assessments, and necessitate defenses to keep them from being exploited—this is the security of machine learning.

In this chapter we introduce some machine learning background and a set of core concepts for both the security of machine learning and machine learning for security which should support the reader in interpreting the findings from later chapters.

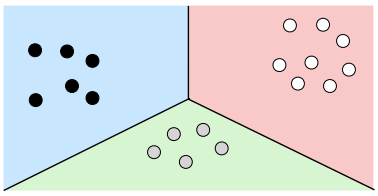
2.1 Fundamentals

We next introduce the concept of the machine learning task and the essential phases that constitute a typical machine learning workflow. We will periodically revisit a *security perspective* to provide intuition for how machine learning is specifically applied to, and affected by, the security domain.

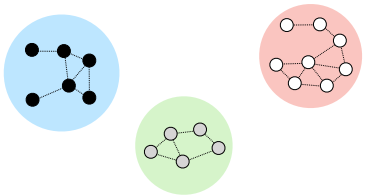
2.1.1 *Machine Learning Tasks*

In the most general sense, machine learning is the process by which algorithms solve tasks using a given dataset without being explicitly programmed how to do so.

Most commonly, these tasks are either *supervised* or *unsupervised*. Supervised tasks are those for which the algorithm is supplied a set of *examples*: *inputs* with corresponding *labels*. Examples of supervised learning tasks are *classification*, in which the target labels are categorical, and *regression*, in which the target labels are numeric. In both cases the algorithm aims to learn a relationship between the inputs and target labels in order to infer labels for unlabeled input at a future date. Figure 2.1(a) shows how the space of examples might be partitioned into decision regions by a supervised learner. In unsupervised tasks, the label information is not provided. A



(a) Supervised classification task



(b) Unsupervised clustering task

Figure 2.1: Examples of class regions and clusters learnt by supervised vs. unsupervised algorithms.

common unsupervised task is *clustering*, in which an algorithm groups inputs together based using some notion of similarity, as illustrated in Figure 2.1(b).

There are different types of machine learning task...

Note that these are not the only categories of task. For example, in *semi-supervised* learning, the algorithm is given a small amount of labeled data combined with a large amount of unlabeled data. This is a promising solution for solving supervised tasks when large quantities of ground truth labels are difficult to obtain.

Security Perspective A classification task in security might be the classification of Twitter messages as either benign or spam [206], whereas an example of clustering would be the familial analysis of malware [90]. Obtaining labels is particularly difficult in security; for example, analyzing whether applications are truly malware is a lengthy, manual process which requires specialist expertise. This makes variants of semi-supervised learning especially attractive. Nevertheless, while many—if not all—algorithms are vulnerable to adversarial interactions to some degree, in this thesis we focus on supervised tasks—and classification specifically—as this is the most common formulation of the security detection problem (*e.g.*, classifying attacks vs. benign examples).

...such as spam detection or malware classification in security.

2.1.2 *Classifier Design and Evaluation*

In the classification task, the dataset consists of n pairs of inputs and their corresponding labels, *e.g.*, $\{ (\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_p, y_p) \}$ where each input $\boldsymbol{x}_i$ is a vector in the space $\mathcal{X} \subseteq \mathbb{R}^r$, each of the r elements is a *feature* describing some attribute of the example, and each label $y \in \mathcal{Y}$ is a categorical value denoting the *ground truth*.

Given this dataset we assume there is some optimal function $g^* : \mathcal{X} \longrightarrow \mathcal{Y}$ which is able to infer the correct label given any $\boldsymbol{x}_i$. While determining g^* is computationally intractable in general, a machine learning classification model, or *classifier*, $g : \mathcal{X} \longrightarrow \mathcal{Y}$ can be used to approximate it.

The classifier builds on a *discriminant function* $h : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ which outputs a real number $h(\boldsymbol{x}, j)$, that represents the fitness of object $\boldsymbol{x}$ to class $j \in \mathcal{Y}$. Higher outputs of the discriminant function h_j represent better fitness to class j, so to convert this score to a label, the class with the maximum score is chosen, *i.e.*, $g(\boldsymbol{x}) = \hat{y} = \arg\max_{j \in \mathcal{Y}} h(\boldsymbol{x}, j)$.

In binary classification, often a single score is output representing the fitness of $\boldsymbol{x}$ to the *positive* class (*e.g.*, the malicious class). A threshold is then used to determine the cutoff between the two classes, which can then be tuned.

Figure 2.2 shows the decision region for a support vector machine (SVM) [66] with a polynomial kernel (used for fitting non-linearly separated data). To approximate g^*, SVMs try to maximize the margin between each class—here the margin is shown by the

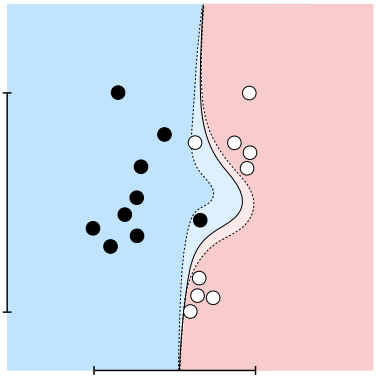


Figure 2.2: Binary SVM classifier showing margins, decision boundary, and class regions.

In classification the goal is to learn a function that distinguishes between classes of data (e.g., malicious vs. benign classes).

dotted line. Between the two margins is the *decision boundary* (solid line); for SVMs the discriminant function $h(\boldsymbol{x})$ represents the distance from this decision boundary to $\boldsymbol{x}$ which will be classified as the corresponding class depending on which side of the decision boundary it falls.

Generalization In order to learn the relationship between $\mathcal{X}$ and $\mathcal{Y}$ and approximate g^*, the dataset is partitioned to create a *training set*, for learning itself, and a *test set*, for evaluation. By partitioning the data, we can ensure that the classifier does not *overfit* to the dataset, because it must *generalize* from the training set to the test set in order to perform well.

To be useful, a model must generalize from training data to unseen test data.

Note that for lab-based evaluations, the test data represents the set of real-world data for which ground truth labels are unknown. Therefore, there is still a risk of tuning the classifier to *overfit* this test set which will inflate our expectations of performance in the wild. To avoid this, the training dataset can be further partitioned into a proper training set and a *validation set* used to evaluate the performance, with the test data being *held out* for a final evaluation which should closely approximate the performance on real, previously-unseen data.

Figure 2.3 shows a learning curve where a model's prediction error is plot as a function of the model complexity (*e.g.*, as training continues over multiple epochs in deep learning). When the model is not complex enough, it *underfits* the training data and increasing the model complexity should improve performance on future validation and test data. When the model is too complex, it can *overfit* the training data, experiencing increasing generalization error on the validation data. The goal is to find the optimal balance between the two, depicted by the dotted line.

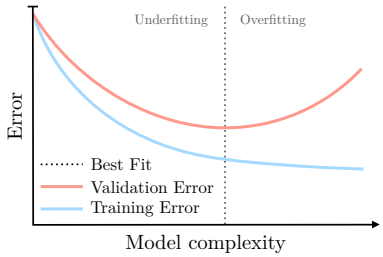


Figure 2.3: Learning curve showing model error as a function of complexity and the resulting under/overfitting.

The assumption underpinning this ability to generalize is that the training, validation, and test sets are all independently drawn from the same joint distribution (*i.i.d.*), and is a core assumption for classification. Furthermore, there is an assumption that the dataset is representative of the true data distribution that will be encountered in the wild.

Inductive Bias Different classification algorithms have different *inductive biases*: the assumptions the classifier makes in order to predict outputs. For example, SVMs assume that different classes can be separated by wide margins, and therefore maximizing the width of this margin when calculating the decision boundary will reduce misclassifications on new data. Similarly, different kernels of SVMs make different assumptions on the separation of the input vectors for each class (*e.g.*, separable by a linear hyperplane, by a polynomial curve, etc) [66]. The set of all valid functions describing the relationships between $\mathcal{X}$ and $\mathcal{Y}$ given these biases is called the *hypothesis space*. During the training process, the classifier produces predicted labels for the training set examples. In order to

Different algorithms make different assumptions about the data...

assess which candidate is best, these predictions are compared to the ground truth labels using a *loss function*, which measures the quantity and/or severity of mistakes the classifier has made. Examples of loss functions are *hinge loss* for support vector machines and *cross-entropy loss* for logistic regression and neural networks.

...minimizing error with different loss functions and different internal and external variables.

During training, the classifier learns a set of internal variables to fit the function g approximating g^*—these are the *parameters* of the model. Additionally there are external variables that affect which candidate function is chosen, which are the *hyperparameters*. Examples of hyperparameters are the values of regularization terms in support vector networks, the learning rate and batch size in neural networks, and the quantity and configuration of weak learners in ensemble methods. Different hyperparameters and decision thresholds are evaluated using performance metrics evaluated on the validation set.

Evaluation To evaluate each candidate on the validation set, and the eventual chosen candidate function g on the test set, a final assessment is performed using some performance metrics. While the loss function can be used, as in training, it is customary to use normalized performance metrics that are easier to interpret and describe specific strengths and weaknesses of the model.

Core to these metrics are the breakdown of correct vs. incorrect predictions. For clarity, here we formulate them with respect to the binary classification class ($\|\mathcal{Y}\| = 2$), but they can extrapolated to multiclass problems. In binary classification, one class is considered the *positive* class, while the other is considered the *negative* class. Therefore, the classifier's predictions can be either correct—true positives (TP) and true negatives (TN)—or incorrect—false positives (FP) and false negatives (FN). From these building blocks we can derive the true positive rate (TPR), the proportion of examples from the positive class which were correctly identified, and the false positive rate (FPR), the proportion of examples from the negative class which were incorrectly identified.

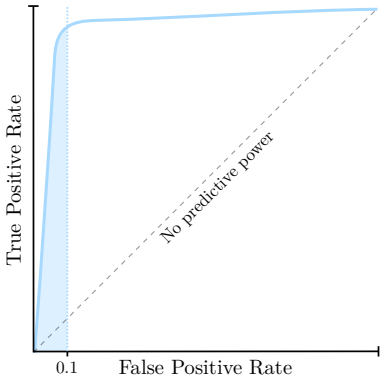


Figure 2.4: Example ROC curve showing AUC at FPR of 0.1. The diagonal acts as a visual reference for AUC = 0.5, which is equivalent to random guessing.

There is an inherent trade-off between these two metrics that can be controlled by tuning the decision score threshold for the positive class. To visualize this tradeoff, a receiver operating characteristic (ROC) curve can be used, as depicted in Figure 2.4. The ROC curve plots TPR over FPR and allows different models to be compared: intuitively a classifier with greater area under the ROC curve (AUC) will have greater performance across more thresholds. While useful, AUC can obscure the true performance when there is a large skew in the class distribution [75].

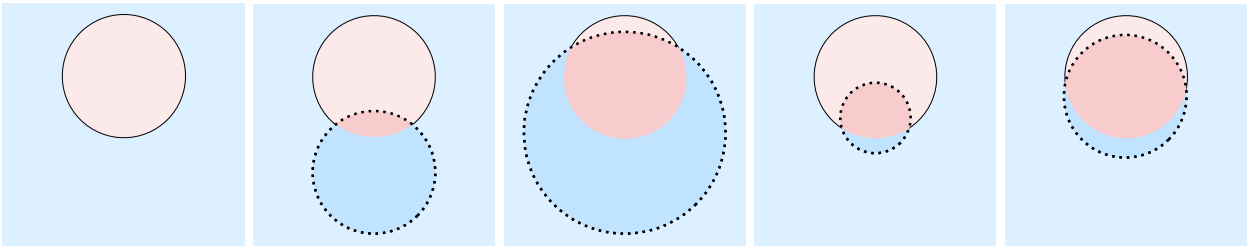
ROC curves can be used to compare the TPR and FPR between classifiers.

As an alternative, Precision and Recall can be used:

$$Precision = \frac{TP}{TP + FP}\ , \tag{2.1}$$

$$Recall = \frac{TP}{TP + FN}\ , \tag{2.2}$$

where Precision denotes the proportion of positive predictions



(a) Positive (red) and negative (blue) examples
(b) Predictions with low Precision, low Recall
(c) Predictions with low Precision, high Recall
(d) Predictions with high Precision, low Recall
(e) Predictions with high Precision, high Recall

Figure 2.5: Illustration of Precision and Recall. Colors denote ground truth for positive (red) and negative (blue) examples, with the positive predicted set shown by the dotted circle, partitioned into TPs and FPs.

which were correct, while Recall captures the proportion of positive examples that were correctly detected. Intuitively, Recall captures how likely it is that no positive examples have been missed, while Precision captures how reliable the positive predictions themselves are. Figure 2.5 illustrates a series of different outcomes and how Precision and Recall can be used to interpret them. In each subfigure, the color depicts the ground truth label of the class, red for the positive class and blue for the negative class. The dotted line encircles the set of examples predicted as the positive class. Graphically we can see clearly how Precision and Recall capture the amount of True Positives (how many examples were detected), False Positives (how many false alarms were raised), and False Negatives (how many examples evaded detection) which are key metrics for evaluating a security detection system.

Alternatively, Precision, Recall, and F_1-Score can be used...

As with TPR and FPR, there is often a tradeoff between Precision and Recall, so the F_1-Score is used to find a balance. The F_1-Score is the harmonic mean of the two scores:

$$F_1\text{-}Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \,. \tag{2.3}$$

Security Perspective Particular care must be taken in the design and evaluation of classifiers for security tasks. Domains in which an adversary has some control over the data distribution are especially common in security. In these settings the i.i.d. assumption will be violated so a defender should take this into account during the evaluation, either by directly assessing robustness of the model using adversarial examples or by simulating the distribution shift that could emerge from adversarial interactions. Similarly, the class distribution must be taken into account as the positive class often has very low prevalence (*e.g.*, attacks compromise an extremely small amount of network traffic [23]). Applying incorrect metrics or misinterpreting their values (as in the *base rate fallacy* [23]) can lead to erroneous conclusions about the effectiveness and robustness of a classifier (see Section 5.10).

...but either way, factors such as the base rate must always be taken into account.

2.1.3 *Before Training: Data Collection*

A dataset will usually consist of raw objects, *e.g.*, images [51], audio [49], programs [231], that are sampled from a *problem space* $\mathcal{Z}$.

In order to derive $\mathcal{X}$, the raw data must first be transformed to a numerical representation [39]. This means the objects in $\mathcal{Z}$ must be transformed into a suitable format for ML processing. We can define a *feature mapping* as a function $\varphi : \mathcal{Z} \longrightarrow \mathcal{X} \subseteq \mathbb{R}^n$ that, given a problem-space object $z \in \mathcal{Z}$, generates an n-dimensional feature vector $\boldsymbol{x} \in \mathcal{X}$, such that $\varphi(z) = \boldsymbol{x}$.

Raw input data, such as apps, must first be mapped to a numerical vector space...

Each value in the feature vector $\boldsymbol{x}$ describes some attribute of the datapoint it represents. What attributes are included or not are decided during the *feature engineering* process. A classic example would in the classification of iris flowers [99]. A set of features such as the sepal and petal lengths are designed, and then measurements for each are taken for each example in the dataset. The procedure is no different for a security task like malware classification. If the presence of certain APIs, registry keys, certificates, or network addresses are deemed significant for determining malicious behavior, they can be used as features. Then a program can be written to automatically extract these features from the set malware examples.

The vector space containing all possible feature vectors is a *feature space*. Typically there is some discrepancy between a feature space and the corresponding space of original raw inputs (*i.e.*, the *problem space*). A simple case is in image recognition tasks, for which the pixels of images are usually represented as real numbers in the continuous range $[0, 1]$. However, to revert these feature vectors back to actual images, they must be discretized and mapped to integers in $[0, 255]$. In practice, this rarely causes issues as the feature space is high fidelity—the feature space and problem space representations are very similar. On the other hand, many security problems suffer from lossy representations; it is still an open problem of how to capture, say, the behavior of an executable, as a feature vector. This difficulty in bridging the feature and problem space representations is an example of a *semantic gap* [263] and is common to other areas of computer security (*e.g.*, unifying low-level instructions and high-level behaviors in program analysis [282]).

...but representing malicious behavior as a high fidelity feature vector is still an open problem....

A unique approach often used in deep learning is *representation learning*, in which features are not explicitly engineered and are instead learned implicitly by the model to produce a *latent feature spaces*. This approach has produced extremely compelling results in computer vision and natural language processing, but its application to security task such as malware are still exploratory [*e.g.*, 232]. While these feature spaces don't require manual engineering and can capture complex relationships between attributes, they may overfit to artifacts and are typically difficult to explain. It is common to consider this implicit feature learning as part of the feature mapping φ.

Security Perspective While we can intuitively recognize that there exists some Platonic ideal [229] of malicious behavior (*i.e.*, 'we know it when we see it'), it is difficult to capture this notion in a

feature space representation. As such, feature engineering plays an important role in the robustness of a model as the semantic gap is often exploited by adversaries who evade detection by intentionally using behaviors that aren't captured by the feature space. A deeper discussion of feature mapping, semantics, and their impact on generating realizable adversarial examples is given in Section 4.3.

...which attackers can abuse to evade detection.

2.1.4 *After Training: Deployment and Operation*

Once the best performing model has been evaluated and chosen, it will be deployed in the wild where the model will produce predictions for previously unseen test inputs. The performance of the model should be tracked over time as gradual changes in the data distribution will cause the performance to decay. Numerous methods exist to help a model adapt to such changes, including incremental learning [13], online learning [69, 203], active learning [250], and classification with rejection [139, 27, 30]. Each approach has its own *cost*, which may or may not be worth the performance improvement depending on the severity of the drift. Given significant performance decay, it may be necessary to completely retrain the model on more recent data, or to revisit the feature engineering and classifier design completely.

Once deployed, models require maintenance to avoid performance decay...

Security Perspective As mentioned previously, while the reasons for dataset shift vary depending on the domain and are not unique to security (*e.g.,* aging faces in facial recognition [200]), they are often exacerbated by adversarial behavior. Importantly, security tasks are susceptible to a kind of *observer effect*. Classifier designers observe the behavior of adversaries in order to design and train the classifier, but once deployed, adversaries will alter their behavior in order to avoid detection (sometimes suddenly and drastically). This feedback loop between defenders and adversaries is often described as a *cat and mouse game* or the *cybersecurity arms race*. An additional threat during deployment comes from data *poisoning*. This attack becomes possible when a model is updated using adversary controlled data, as can occur with many of the previously mentioned adaptation strategies. By poisoning the data, attackers can reduce the effectiveness of the model [36], or even inject backdoors which cause the model to produce specific outputs when triggered with specially crafted inputs [121, 252].

...as their presence alone will cause attacker behavior to change, resulting in drift.

2.2 Core Concepts in the Security of Machine Learning

The most common attacks against machine learning algorithms can be categorized as either test-time attacks (*e.g.,* evasion), training time attacks (*e.g.,* poisoning and backdoors), or privacy attacks (*e.g.,* model stealing and membership inference). In this thesis we

focus primarily on test-time evasion attacks, however, it is useful to contextualize these among the broader threat landscape, as many concepts discussed may be extrapolated to other settings. For example, Chapter 4 explores how to generate realizable adversarial inputs in the problem-space for targeting malware detectors, *i.e.*, functioning malware that maps to an adversarial feature vector. We approach this problem from the perspective of evasion attacks, but generating such problem-space apps is also a necessary step for poisoning or trojaning malware detectors.

Machine learning algorithms have their own set of vulnerabilities...

2.2.1 *Adversarial Examples*

In information security the moniker *adversarial examples* may seem redundant, as adversarial inputs are ubiquitous. For as long as there have been defenses, there have been attempts to evade them, and as we have discussed, in detection tasks the data of interest is inherently adversarial (*e.g.*, malicious apps or attack traffic).

Nevertheless, the term originates from the machine learning field [279] to more specifically describe points which are intentionally crafted by an attacker and which are misclassified by learning algorithms in a counterintuitive way. What is counterintuitive about them is that the examples can be extremely similar to non-adversarial inputs. For example, an image of a cat may be misclassified as a dog, despite looking visually identical (to the human eye at least) to another image of a cat that is correctly classified. To extend the analogy to a security task, two apps may exhibit the exact same malicious behavior, but one will be classified correctly and the other will evade detection.

Figure 2.6 illustrates how this phenomenon can arise for toy data and a linear classifier. The squares around each example demarcate some L_∞ norm distance, within which the set of examples are similar and share a ground truth label. For example, for a set of images in the image recognition task, the L_∞ norm may be small enough that any differences between examples of the set are visually imperceptible. As is clear from the illustration, there will be examples in these sets which are also in the *error set*, *i.e.*, examples which fall on the other side of the decision boundary and are misclassified. Two such examples are marked by the stars ($*$).

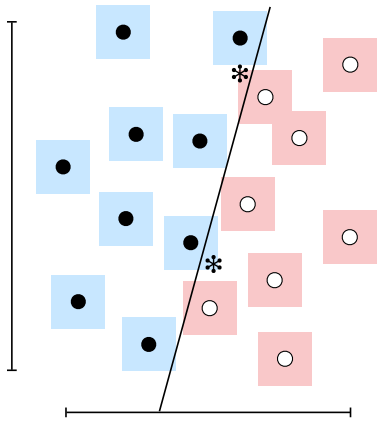


Figure 2.6: Illustration of adversarial examples ($*$) for a linear binary classifier and given L_∞ constraint represented by the colored squares.

2.2.2 *Test-Time Attacks*

Adversarial examples can be crafted to perform test-time *evasion attacks*, the archetypal attack against machine learning classifiers in which an input is perturbed in order to fool the target model [279, 85, 34, 226]. Evasion attacks can be *targeted*—or class-dependent—if they aim to flip the prediction to a specific class, or *untargeted* if they aim to cause a misclassification regardless of the final class (these are equivalent in binary classification) [52].

...such as the presence of adversarial examples that can evade detection while retaining malicious functionality.

Figure 2.7 depicts a gradient-driven evasion attack in which an example from class ○ is perturbed to cross the decision boundary where it will be misclassified. The dotted circle represents a constraint on the L_2 norm used to ensure the perturbed example is still a plausible member of class ○. The contours show the values of a loss function quantifying the fitness of the example to class ● from dark red (high loss) to dark blue (low loss).

The dotted arrow shows small steps taken iteratively in the direction of the negative gradient of the loss with respect to the example which can be followed to find a perturbation with which the example will be misclassified as class ● with high confidence. Algorithms that use gradient-driven optimizations of this sort include the fast gradient sign method (FGSM) [112], Carlini–Wagner (CW) [51], the basic iterative method (BIM) [160], and projected gradient descent (PGD) [181].

When a perturbation is generated to make a single input evasive, it is termed an *input-specific perturbation*. The alternative is to generate a *universal adversarial perturbation* (UAP) which can applied to multiple inputs to make all of them evasive [198, 304]. UAPs significantly reduce the effort for an attacker to create adversarial examples, enabling practical and realistic attacks across different applications, and are useful for revealing systemic vulnerabilities in the model [64, 304].

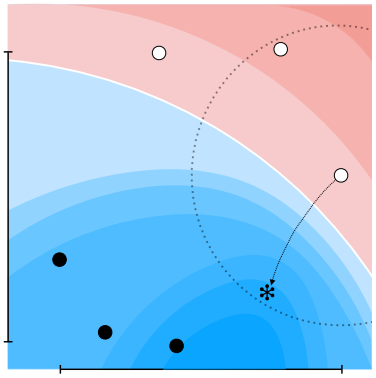


Figure 2.7: Evasion attack showing the path taken to find an adversarial variant ($*$) of an example from class ○ given an L_2 constraint (dotted circle).

2.2.3 *Training-Time Attacks*

Attackers can poison the training process to degrade performance...

Attacks can also be performed at training time. In these threat models, the attacker is assumed to have partial control over some combination of: the training data, the labeling of some training data, the parameters of the model. An example in security would be where an antivirus company retrains their model using recent malware submissions; clearly the attacker partially controls this data [251]. In some extreme threat models, the attacker controls *the entire training process and distribution* of a model (part of the *model supply chain* [121]. This is particularly relevant where extremely large vision or language models are pretrained by the few entities with the data and resources to do so (corporations such as Google and OpenAI) and released to practitioners to fine-tune on their own tasks. It is possible that these models have been generated such that their outputs are not wholly trustworthy.

The foundational training time attack is *poisoning*. In poisoning, a small fraction of training data is perturbed or mislabeled in order to maximize model error at test time [36]. The term 'poisoning attack' usually refers to an attack against the availability of a model, *i.e.*, where the only goal is to maximize model error as a denial of service. However the mechanism of poisoning training data also facilitates more insidious attacks such as the *backdoor*, or *trojan*, attack [*e.g.*, 121, 244, 252].

In backdoor attacks, training data is poisoned in a very targeted way, by applying a specific pattern of features, called the *trigger*, to the poisoned examples. At test time, only inputs with the trigger will be misclassified while other *clean* inputs are unaffected. Usually the data is poisoned in such a way that a backdoor is inserted from one class (or many classes) to another, for example in malware detection it would not be advantageous to insert a backdoor from the benign to the malicious class, only from the malicious to the benign class.

...or even insert backdoors to activate at test time.

Figure 2.8 illustrates a type of backdoor attack called a *clean-label attack* in which the attacker can poison the data but not control the labeling [252]. This means they must ensure the poisoned data remains a plausible member of the ground truth set. In the example, the dotted line depicts the decision boundary before poisoning, the solid line the decision boundary after poisoning, and the two center top examples with tails represent triggered examples. The triggered example of class ● is the poisoned example. Including it forces the model to fit to it and alters the decision boundary—introducing the backdoor. Now new test-time examples, such as the triggered example of class ○, can be presented to the model and will be misclassified.

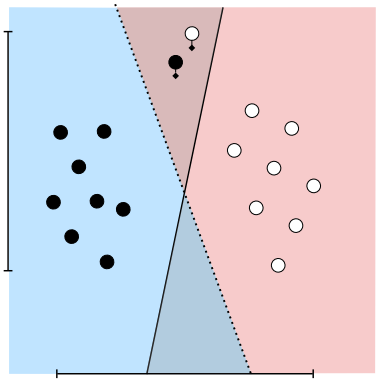


Figure 2.8: Clean-label attack introducing a backdoor shown by the change in decision boundary (from dotted to unbroken line). Tails indicate the triggered examples: a training example (●) used to introduce the backdoor and a test example (○) which is now misclassified.

It is important to note that backdoor attacks can also be performed without data poisoning, usually by directly altering the parameters of a trained model or the code used to generate it [243]. Such attacks have a smaller footprint but require greater control over the model and are often associated with insider threats.

2.2.4 *Privacy Attacks*

In privacy attacks the attacker tries to extract information about the training data used to train the model or the parameters of the model itself. These attacks are particularly relevant to cloud-based Machine-Learning-as-a-Service (MLaaS) providers where the attacker only has black-box access and where the models may be trained with private user data or themselves represent valuable intellectual property.

In *membership inference attacks* the goal is to determine whether a given example was a member of the training set or not. This can itself be formulated as a classification problem, in which the attacker generates shadow models trained with different training sets (where the membership of each example is known) and then uses these models to train a metaclassifier to distinguish between outputs of models that do or do not contain a particular example [258].

Attacks can force the model to leak private information about the training data...

Similarly in *model inversion* attacks, the attacker is able to infer information about the training data from the model outputs [103, 102]. Unlike in membership inference, model inversion attacks cannot say whether an individual example was present in the training set, but instead can extract an average representation of the training

inputs for a particular class. For example, in biometric applications such as facial recognition where each class is an individual, the attacker can reconstruct the average features of a specific person [102].

The extraction of training data usually exploits a tendency of models to *memorize* training examples (a form of overfitting). A clear illustration of this is in attacks against large generative language models, in which the model can be stimulated to output long sequences of private training data such as addresses, phone numbers, and social security numbers [53].

...or even steal the parameters of the model itself.

In more of an attack on model secrecy rather than privacy, *model stealing* attacks try to extract the parameters of the model itself. A typical strategy is to query the target model to produce a set of input–output pairs, and then solve for the parameters in order to replicate the behavior of the target model [290]. Models reconstructed in this way can also act as a surrogate model for test- and training-time attacks or act as an intermediate step towards recovering information about the training data.

2.2.5 *Transferability*

Transferability is the property by which attacks against one machine learning model can succeed against another model trained for the same task, even if they have different architectures, use different learning algorithms, or were trained on different datasets [78, 214]. Transferability has wide-ranging implications for machine learning security, because it allows attackers to craft strong generalizable attacks while having limited knowledge of the target model. For example, an attacker can use their own set of *surrogate data* (ideally sampled from the same distribution as the target model's training set) to train a local *surrogate model*. They could then use white-box, gradient-driven methods to generate examples which evade the surrogate model, but which would *transfer to*, and evade, the target model. They may also be able to reuse examples to evade multiple different models, reducing the cost it takes to generate them.

Transferability allows approximations of white-box attacks to be applied in black-box settings.

2.3 Core Concepts in Machine Learning for Security

In this section we outline some core concepts used in the evaluation of security detectors based on machine learning. Threat models are vitally important for defining the scope of an attack or defense. Closely tied to the threat model are notions of cost, robustness, and adaptive worst-case attackers, all of which have implications for the evaluation of defenses.

2.3.1 *Threat Models*

Threat models are a vital component of all discussions on security as they define the scope of an attack or defense. Claims about applicability, strength, and cost of attacks, and about the effectiveness, robustness, and generalizability of defenses, must all be interpreted with respect to a given threat model; that is, without a threat model, such claims are not falsifiable.

Threat models describe...

While there are many ways to formulate a threat model, here we give an overview of the threat model formalization proposed in Biggio and Roli [34] which is used throughout this thesis. This formulation frames the threat model in terms of *attacker knowledge* (what they know about the model and its defenses) and *attacker capability* (their ability to act on this knowledge).

Attacker Knowledge We represent the knowledge as a set $\theta \in \Theta$ which may contain (i) training data $\mathcal{D}$, (ii) the feature space $\mathcal{X}$, (iii) the learning algorithm g, along with the loss function $\mathcal{L}$ minimized during training, (iv) the model parameters/hyperparameters $\boldsymbol{w}$. A parameter is marked with a *hat* symbol if the attacker knowledge of it is limited or only an estimate (i.e., $\hat{\mathcal{D}}$, $\hat{\mathcal{X}}$, $\hat{g}$, $\hat{\boldsymbol{w}}$). There are three major scenarios [34]:

...what an attacker knows about the model...

- *Perfect Knowledge (PK) white-box attacks*, in which the attacker knows all parameters and $\theta_{PK} = (\mathcal{D}, \mathcal{X}, g, \boldsymbol{w})$.
- *Limited Knowledge (LK) gray-box attacks*, in which the attacker has some knowledge on the target system. Two common settings are LK with Surrogate Data (LK-SD), where $\theta_{LK-SD} = (\hat{\mathcal{D}}, \mathcal{X}, g, \hat{\boldsymbol{w}})$, and LK with Surrogate Learners, where $\theta_{LK-SL} = (\hat{\mathcal{D}}, \mathcal{X}, \hat{g}, \hat{\boldsymbol{w}})$. Knowledge of the feature space and the ability to collect surrogate data, $\theta \supseteq (\hat{\mathcal{D}}, \mathcal{X})$, enables the attacker to perform *mimicry attacks* in which the attacker manipulates examples to resemble the high density region of the target class [37, 100].
- *Zero Knowledge (ZK) black-box attacks*, where the attacker has no information on the target system, but has some information on which kind of feature extraction is performed (e.g., only static analysis in programs, or structural features in PDFs). In this case, $\theta_{ZK} = (\hat{\mathcal{D}}, \hat{\mathcal{X}}, \hat{g}, \hat{\boldsymbol{w}})$.

Note that θ_{PK} and θ_{LK} imply knowledge of any defenses used to secure the target system against adversarial examples, depending on the degree to which each element is known [50].

Attacker Capability The capability of an attacker is expressed in terms of their ability to modify values in the feature space $\mathcal{X}$, as well as the corresponding problem space $\mathcal{Z}$ which determines their ability to generate effective real-world attacks. Formally, we can define attacker capability as a set of feature-space constraints Ω

and problem-space constraints Γ, which we elaborate on further in Section 4.3.

...and their capability to act on that knowledge.

Note that the attacker's knowledge and capability can also be expressed according to the FAIL [277] model as follows: knowledge of *Features* $\mathcal{X}$ (F), the learning *Algorithm* g (A), *Instances* in training $\mathcal{D}$ (I), *Leverage* on feature space and problem space with Ω and Γ (L). More details on threat models formulations can be found in [34, 277].

2.3.2 *Attacker Cost vs. Capability*

Defending against adversarial examples is an extremely difficult task, and it may be that a technical solution that mitigates their effect completely is not possible without a significant paradigm shift. Essentially, when models misclassify adversarial examples, they are doing what they are supposed to do—exploiting highly predictive features in the dataset to discriminate between classes. The real issue is that this behavior does not match up with our intuitions, nor with how humans behave given the same task [132].

'Raising the bar' involves increasing the attacker's cost for invoking a given capability...

An intermediate step towards complete mitigation is to *raise the bar* for the attacker, to provide defenses which force the attacker to expend much greater cost in order to exercise the same capability as before. Increasing attacker cost reduces harm as attackers are not able to iterate on their attacks as quickly, must choose fewer targets, and may be forced to produce weaker or more complex attacks which are then easier to detect themselves.

2.3.3 *Robustness*

Closely tied to the threat model is *robustness*, which is usually defined with respect to some bounded adversarial capability. For example, in the computer vision domain, an attacker can perturb the pixels of an image in order to produce an adversarial example that is misclassified. We can quantify the size of the perturbation the attacker adds to the image using a distance such as the L_2 or L_∞ norms. Typically, smaller perturbations will be less noticeable as the image still closely resembles the original, while larger perturbations will produce more evasive examples. At the extreme, the attacker can completely transform the image to look like an image from another class, but at this point the image has lost all semantic relevance to its original class. As such, it usually doesn't make sense to consider an unconstrained attacker and instead we set some reasonable bound on the size of the perturbation. Given such a constraint, we can define robustness as the worst-case performance of the model given this adversary capability.

...and 'robustness' quantifies the height of the bar, assessing how resilient a model is under attack.

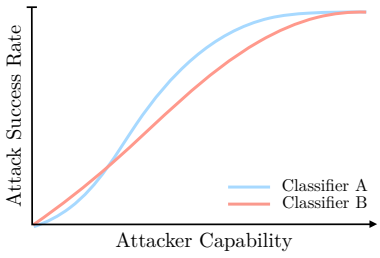


Figure 2.9: Security evaluation curve comparing robustness between two classifiers.

Figure 2.9 depicts a security evaluation curve which can be used to visualize this relationship. The curve shows the success rate of an attack as a function of the attacker's capability (*e.g.*, the pertur-

bation budget in the computer vision example). The robustness of different models can be compared by these curves, for example the figure suggests that overall Classifier B is more robust than Classifier A against medium-strength attacks.

In this thesis we will also use robustness to refer to robustness against concept drift in relative terms. We say model *a* is more robust than model *b* if model *a* has greater detection performance given the same severity of drift, or that model *a* is more robust than model *b* if model *a* has the same detection performance given more severe drift.

2.3.4 *Security by Obscurity and Adaptive Attackers*

Security by obscurity is a defense mechanism in which parts of the design or implementation of a system are kept secret in order to prevent them from being exploited. While it may have value as part of defense-in-depth, it cannot be relied upon as attackers can often find ways to infer or leak the information being hidden. In contrast, *Kerckhoff's principle* is a cryptographic design principle which holds that a system should be secure even if the attacker knows everything about the system—except the secret key [147]. This notion can also be applied to the evaluation of machine learning-based systems.

Defenses should not rely on security by obscurity...

A classic example in machine learning security is the use of *gradient masking*, a category of defenses in which a model does not produce useful gradients [*e.g.,* 280, 120, 215]. As gradients are often used to guide optimization-based algorithms for crafting adversarial examples [*e.g.,* 112], it was suggested that these models would be more robust. However, an attacker aware of this defense would be motivated to recover usable gradients, which is possible via black-box stimulation of the model [216] or by using an approximation of the model which *is* differentiable [22].

An attacker who has perfect knowledge of the system, its defenses, *and* attempts to exploit them, is an *adaptive attacker* [52] and must be considered in threat models for evaluating defenses.

...and should be evaluated assuming worst-case attackers.

2.4 Summary

The reader should now be comfortable with the machine learning classification task and how classifiers can be applied to binary security detection tasks (*e.g.,* classifying malware vs. goodware or attack vs. benign traffic). They should be familiar with the presence of adversarial examples and different types of attacks against machine learning algorithms; typical metrics for quantifying the success of a classifier such as AUC, Precision, Recall, F_1-Score, and robustness; and threat models used to define the scope of attacks or defenses with respect to attacker knowledge and capability.

Part II:

Adversarial Interactions

> Rogues are very keen in their profession, and know already much more than we can teach them.
>
> Alfred Charles Hobbs

3 *Characterizing Concept Drift in Security*

To provide a characterization of the hostile environment, this part discusses two aspects of adversarial interactions with security detectors: concept drift and realizable adversarial examples. We first give an overview of concept drift, outlining the different forms that concept drift takes and how they manifest in security data. In particular we illustrate that concept drift and adversarial inputs are inextricably related in security detection tasks and motivate why they should be studied in tandem.

Concept drift is a central motif throughout this thesis as it is through concept drift that we indirectly observe the effects of the hostile environment. Chapter 4 will discuss the constituent parts of concept drift in security: realizable adversarial examples; Chapter 5 will discuss ways of performing evaluations in a drifting setting and measuring the performance decay induced by concept drift; and Chapter 6 will propose a methodology for identifying and quarantining drifting examples.

3.1 Key Insights

For reference, this chapter provides the following contributions:

- We first provide an overview of the different types of concept drift and frame them in the context of security detection tasks. We posit that concept drift in security should be treated practically as an artificial process and largely exists as an emergent phenomenon arising from the presence of adversarial inputs (Section 3.2).

- We then propose a unifying theoretical framework to describe covariate shift, concept drift, and the space of adversarial examples, by modeling them as submanifolds and morphisms of the lower dimensional class manifold. This framework supports the hypothesis that the three phenomena are different aspects of the same underlying issue, motivating robustness research that jointly tackles all three (Section 3.3).

3.2 Overview

One of the greatest challenges facing machine learning-based security detectors is the presence of *dataset shift* [144, 194, 6, 139, 13] as the distribution of the malicious class at test time (*i.e.*, at deployment) begins to diverge from the training distribution. This violates one of the core assumptions of most classification algorithms: that the training and test time examples are identically and independently drawn from the same joint distribution (i.i.d.). As this assumption weakens over time, the classifier's predictions become less and less reliable and performance degrades.

Concept drift describes changes in the data distribution over time...

Dataset shift can be broadly categorized into three types of shift [199]. *Covariate shift* refers to a change in the distribution of $P(\boldsymbol{x} \in \mathcal{X})$, when the frequency of certain features rises or falls (*e.g.*, variations in API call frequencies or traffic statistics over time). *Prior probability shift* or *label shift* is a change in the distribution of $P(y \in \mathcal{Y})$, when the base rate of a particular class is altered (*e.g.*, an increase in attacks or malware prevalence over time). *Concept drift* is a change in the distribution $P(y \in \mathcal{Y} | \boldsymbol{x} \in \mathcal{X})$. This often occurs when the definition of the ground truth changes, for example, if a new family of malware arises which, given the feature space representation $\mathcal{X}$, is indistinguishable from benign applications. Due to the model's limited knowledge, the model will start misclassifying examples from the new family, even if no covariate or prior probability shift has occurred.

With some notable exceptions [*e.g.*, 144, 193], it is common for concept drift to be treated as a 'natural' evolution in the class distributions over time [187, 316, 325, 141] in which concept drift is treated as unrelated to adversarial behavior. This is certainly the case for domains outside of security, *e.g.*, in the problem of aging faces for facial recognition [201].

However, we posit that this perspective is fallacious in the security domain.

While there are natural sources of gradual drift in security such as changes in market trends, user preferences, or new developer APIs affecting malware classification [325], it is intuitive that drift in security data is largely driven artificially by drift in the malicious class as a result of the hostile environment [144]. For example, the impetus for concept drift in malware classification is that malware authors are driven by the profit motive to try and evade detection or classification by app store owners, antivirus companies, and users. This incentivizes them to innovate: to obfuscate features of their malware, develop new methods for exploitation and persistence, and explore new avenues of profiteering and abuse. This causes the definition of malware to evolve over time, sometimes in drastic or unexpected ways [283]. The same is true in modern network traffic environments where attacks are polymorphic and continuously evolving in order to adapt to defenses [100]. Fig-

...which in security contexts largely exists in the malicious class and is driven by adversarial behavior.

ure 3.1 illustrates this as a function of performance over time. At a fine-granular level, adversarial attacks cause immediate drops in classification performance which, over time, accumulate in a steady downward trend as adversaries continue to adapt. This cumulative effect is what we perceive as concept drift.

This intuition is supported empirically. In Chapter 5 we plot the performance over time for three Android malware classifiers [19, 187, 118] with respect to both the benign and malicious class and observe performance decay which is negligible in the benign class but significant in the malicious class. In Chapter 6 we perform similar experiments to identify drifting examples for malware detectors from the Android [19], Windows PE [11], and PDF [271] domains and report similar trends.

In this chapter, we take a different approach. Inspired by past work on the geometry of adversarial examples [*e.g.*, 150, 110], we propose a theoretical framework that provides a unifying perspective on the phenomena of adversarial examples, covariate shift, and concept drift. This provides further support to our hypothesis that in security, these notions are describing different aspects of the same underlying issue—adversarial interactions with the model. This motivates research that considers the issues jointly, as defenses against one will inform the design of defenses against the others.

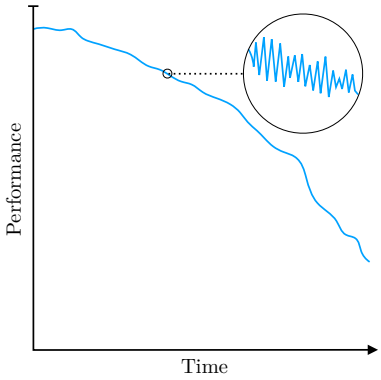


Figure 3.1: Illustration of cumulative performance decay induced by concept drift resulting from ongoing adversarial interactions.

Later in this thesis we provide some empirical evidence to support this idea...

> Note that in practice, it can be extremely difficult to determine how much error should be attributed to each type of shift [199]. Given this, it is common in the security community to collectively refer to all types of shift as *concept drift*. Throughout this thesis we will continue this custom—with the exception of the remainder of this chapter in which we specifically deal with covariate shift and concept drift as separate phenomena.

3.3 Unifying Drifting and Adversarial Examples

In this section we propose a unifying framework for describing covariate shift, concept drift, and adversarial examples which suggests that they are simply different manifestations of the same adversarial behavior. This further implies that reasoning about one may inform defenses against the others, motivating research which jointly considers all three phenomena.

...while this chapter offers some theoretical reasoning to better understand the relationship between adversarial examples and drift.

3.3.1 *Modeling Drift Using Manifolds*

The manifold hypothesis states that real-world natural data embedded in a high-dimensional space is actually concentrated around a manifold of much lower dimensionality [94, 32] and has been cited as the reason for machine learning problems being tractable [150].

While attempts have been made to prove the hypothesis under certain strict conditions [*e.g.*, 94], it is difficult to show it to be universally true for complex data. However, practically it remains a useful heuristic and has been highly influential as a prior exploited by representation learning methods, for example in principal components analysis (PCA) [130, 220], sparse coding algorithms [207, 322], and the development of autoencoders [237, 299, 129].

We begin by modeling the classes as manifolds and submanifolds...

In this chapter, we will assume that the manifold hypothesis is accurate for describing security data and use it to unify the notions of adversarial examples and concept drift in a hostile environment, which will provide some intuition to support how these issues can be jointly mitigated.

Definition 1 (Class Manifold) *We assume data from each class is sampled from a k dimensional manifold denoted as* $\mathcal{M}_j \subset \mathbb{R}^d$ *for each class* $j \in \mathcal{Y}$.

We assume that the manifold defines the class sufficiently such that they can be interpreted as the fundamental ground truth of each class.

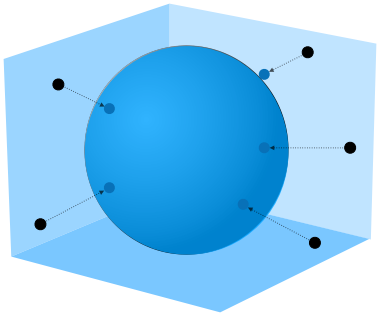


Figure 3.2: Illustration of the manifold hypothesis showing data embedded in a 3D space projected to corresponding points concentrated on or near a 2D manifold.

Definition 2 (Perceived Manifold) *At any point in time t we perceive a submanifold* $\mathcal{P}_j^t \subseteq \mathcal{M}_j^t$ *of the class manifold for each class* $j \in \mathcal{Y}$.

The perceived manifold arises from the fact that, in practice, we are only exposed to a subpopulation of the class distribution. For example, consider a malware detector *Det* deployed over a week-long period. At any point in time the malware encountered will only be a subset of the total population of malware—these points are sampled from the perceived manifold.

Definition 3 (Concept Drift) *As concept drift is a change in the class definition itself, which we assume is geometrically represented by the class manifold, we can define concept drift as a morphism* φ_d*:*

$$\varphi_d : \mathcal{M}_j^t \longrightarrow \mathcal{M}_j^{t^*} . \tag{3.1}$$

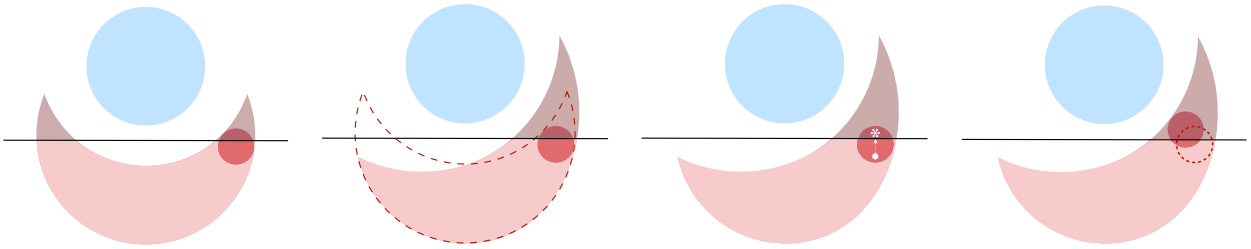
Following from the previous example, during the week period, malware authors may discover that a set of behavior that was previously considered benign can be exploited for malicious functionality. As they adopt this technique, *Det* will produce errors as the definition of what malware is expands to include apps that were previously considered as goodware.

...and define covariate shift and concept drift as morphisms of the manifolds over time...

Definition 4 (Covariate Shift) *We can define covariate shift as a morphism* φ_c *of the perceived manifold within the class manifold, where* $\mathcal{M}_j^t \subseteq \mathcal{M}_j^{t^*}$*, depending on the presence of concept drift:*

$$\varphi_c : \mathcal{P}_j^t \subseteq \mathcal{M}_j^t \longrightarrow \mathcal{P}_j^{t^*} \subseteq \mathcal{M}_j^{t^*} . \tag{3.2}$$

This definition is intuitive following the definition and example for the perceived manifold. For *Det*, as it encounters new data from



(a) Class and malicious perceived manifolds

(b) Concept drift affecting the malicious class manifold

(c) Adversarial perturbation inducing a misclassification

(d) Covariate shift affecting malicious perceived manifold

Figure 3.3: Illustration of drift phenomena showing class manifolds $\mathcal{M}_0$ (blue) and $\mathcal{M}_1$ (red), and perceived manifold $\mathcal{P}_1$ (red circle). Darker areas depict the error set for each manifold. Dotted lines show manifold before drift occurred. White circle and star show adversarial point before and after perturbation, respectively.

one time interval to the next, it is likely that the feature distribution will change as adversarial activity fluctuates. This activity manifests as covariate shift. Note that covariate shift and concept drift can occur simultaneously where both the perceived manifold and the class manifold change over time.

Definition 5 (Adversarial Example) *We define an error set $E(\mathcal{M}_j^t)$ as the set of inputs sampled from the class manifold $\mathcal{M}_j^t$ that are misclassified by the model. In this context we can define an adversarial example derived from $\boldsymbol{x} \in \mathcal{P}_j^t$ as a point $\boldsymbol{x}' \in \mathcal{M}_j^t$ such that $\boldsymbol{x}' \in E(\mathcal{M}_j^t)$.*

While practically the adversary will seek to find the smallest distance between $\boldsymbol{x}$ and $\boldsymbol{x}'$ to fulfill constraints on plausibility and inconspicuousness (*e.g.*, the point in the error set closest to $\boldsymbol{x}$), here we loosen the definition to include any point that can be intentionally crafted by an attacker to induce error in the model. Note that adversarial attacks are by definition dependent on the decision boundary of the model, which is not true for the other two phenomena.

...while adversarial examples exist as members of an error set concentrated near the class manifold.

3.3.2 *Implications for Security Detection*

Having provided a geometric formulation for concept drift, covariate shift, and adversarial examples, some relationships between the three phenomena become apparent which can help us gain a deeper intuition on their nature. Here we discuss some of the implications, framed for a security detection task with a fixed model (*i.e.*, not retrained over time) where $\mathcal{Y} = \{0, 1\}$, denoting the benign and malicious class, respectively. Three observations in particular suggest that the three issues are simply different facets of the same phenomena in such a setting:

This modeling reveals some relationships between the phenomena:...

Concept drift and the adversarial space depend on one another. The space of possible adversarial points in a binary security detection task at any point in time coincides with the error set $E(\mathcal{M}_1^t)$, this is immediate by the definition. As concept drift transforms $\mathcal{M}_1^t$ it is highly likely that the volume of this error set, $Vol_E(\mathcal{M}_1^t)$, will also change, potentially increasing or decreasing the adversarial space—that is, concept drift dictates the adversarial space. This

...that concept dictates the size of the adversarial space...

demonstrates a very intimate relationship between concept drift and adversarial examples. We posit that this relationship exists due to concept drift being artificially driven by adversarial behavior (*e.g.,* attackers searching for adversarial examples) and therefore in practice concept drift likely *increases* the volume of the error set.

Covariate shift and adversarial examples share an attack space. Note that there exists a union of perceived manifolds that is a cover of $\mathcal{M}_1^t$ and thus also spans $E(\mathcal{M}_1^t)$. Intuitively we can see why this should be the case by visualizing the perceived manifold 'move' across the class manifold over time as adversaries perturb the features of inputs in order to craft adversarial examples. In an extreme case in which adversaries discover all possible adversarial examples that lie on the class manifold (*i.e.,* without inducing concept drift), we would perceive this as a covariate shift mapping $\mathcal{P}_1^t$ to a cover of $E(\mathcal{M}_1^{t^*})$. This shows that the potential threat posed by both phenomena is equivalent.

...that covariate shift and adversarial examples share a potential attack space...

Robustness against all three phenomena can be improved jointly. We can formulate robustness as an optimization problem that aims to minimize the volume of the adversarial space across all classes:

$$\min_{\theta} \sum_{j \in Y} Vol_E(M_j^t)\,, \tag{3.3}$$

where θ represents the parameters of the model and any applied defenses.

We can imagine a defense against adversarial examples and the effects of covariate shift applied at time t which is able to successfully reduce this volume at time t^* such that:

$$\sum_{j \in Y} Vol_E(M_j^{t^*}) = \sum_{j \in Y} Vol_E(M_j^t) - Def(M_j^t)\,. \tag{3.4}$$

...and that a single defense can jointly mitigate all three issues.

However, concept drift can morph the class manifold, scaling the volume of the error set by a factor of $\gamma_j \in [0, \infty)$. Additionally, if our defense only hardens a particular region of the space (*e.g.,* L_2 norm balls around points sampled from the manifold at time t), then concept drift can render part of the defense ineffective. In this case only the intersection of the previously defended region and the new class manifold is useful:

$$\sum_{j \in Y} Vol_E(M_j^{t^*}) = \sum_{j \in Y} Vol_E(\varphi_d(\mathcal{M}_j^t) - Def(\mathcal{M}_j^t) \cap \varphi_d(\mathcal{M}_j^t)))\,. \tag{3.5}$$

In both cases, we are practically only able to control the impact of the defensive function, in which case improving robustness against all phenomena can be achieved by improving the space covered by the defense Def—although to different degrees.

3.3.3 *Implications for Robust Feature Space Design*

It is worth briefly taking a moment to discuss the design of robust feature spaces in the context of these three phenomena. Concept drift always occurs relative to a particular feature space—the fidelity of the feature representation affects the ability for adversaries to exploit the *semantic gap*: the gap between the semantics captured by the feature space and those present in the original objects [263]. For example, consider a statically extracted feature space in malware detection. If a new exploitable API is added to the target platform which is not included in the feature space, malware authors may include it in their apps without it being visible to the model. Designing feature spaces which accurately capture all malicious properties and behaviors has proven extremely challenging and remains an open problem. However despite this, we intuitively understand that there is some Platonic ideal [229] of malicious behavior—*i.e.,* we know it when we see it. Removing anything from this ideal definition would make it no longer apply to malicious examples, and adding anything more to it would be redundant.

Manifold modeling tells us something about the ideal dimensionality of robust feature spaces...

These notions can be interpreted more practically when viewed from the perspective of manifolds. Khoury and Hadfield-Menell [150] assert that a feature space of too high dimensionality (specifically as the codimension $d - k$ increases) becomes less robust as there are an increasing number of directions off the class manifold in which to craft adversarial perturbations. As concept drift is a phenomenon driven by adversarial activity, this assertion can be extended to concept drift and covariate shift as well.

...high codimension of the class manifold within the higher level feature space reduces robustness...

The question then is by how much can the codimension be reduced. The *intrinsic dimensionality* describes the minimal number of dimensions needed to describe any member of the class manifold, in contrast to an extrinsic, or ambient, dimensionality that it can be embedded within [94]. A feature space with too low dimensionality will mean that there are certain regions of space in which the classes overlap and cannot be well-separated. When this occurs there will be feature vectors for which it is not possible to determine which class it belongs too, so more features must be added.

...but too low dimensionality will increase entanglement between classes, causing classification errors.

3.4 Related Work

Geometry of Adversarial Examples Khoury and Hadfield-Menell [150] propose a geometric framework to analyze the high-dimensional geometry of adversarial examples. Their framework reveals the effect of codimension on robustness, as well as how the use of different norms affect robustness evaluations. Similar to our work, they build upon the manifold hypothesis, however we extend this modeling to concept drift and covariate shift as well. Gilmer et al. [110] use geometric reasoning to illustrate a relationship between adversarial examples and images corrupted by additive

Gaussian noise. They observe that training procedures to improve robustness against adversarial examples can improve corruption robustness and that can training using Gaussian noise can moderately improve robustness against adversarial examples. This work inspires us to reason about similar relationships between adversarial examples and adversarial drift.

Characterizations of Drift Moreno-Torres et al. [199] provide an excellent general taxonomy of dataset shift and we defer to their terminology in this thesis. Gama et al. [108] describe types of concept drift in the context of online learning, however they focus on applications such as recommendation systems and do not consider adversarial sources of drift. In contrast, Kantchelian et al. [144] position covariate shift as adversarially driven and outline practical research directions to help design systems which can respond to these changes. Sehwag et al. [248, 247] investigate adversarial i.i.d. violations from a different perspective, studying adversarial examples derived from out-of-distribution inputs. They find that existing out-of-distribution detectors and adversarial examples detectors are insufficient to detect them and urge the inclusion of such attacks in the design of future defenses.

3.5 Summary

In this chapter we have characterized the presence of concept drift for security detection tasks. We have further illustrated an intrinsic relationship between covariate shift, concept drift, and the space of adversarial examples through a theoretical framework built on the manifold hypothesis. These relationships suggests that the three phenomena are three aspects of the same underlying issue, motivating research that studies them jointly.

4 *Realizable Adversarial Attacks in Security*

A DRIVING FORCE OF CONCEPT DRIFT in security is the presence of adversarial examples. To stay effective and maintain profitability, attackers seek to discover adversarial examples which are misclassified by the machine learning-based detector while still retaining their malicious functionality. The more effective these examples are, the more severe the resulting concept drift will be.

However, generating realizable adversarial attacks in security domains by applying powerful, gradient-driven, state-of-the-art attacks, such as those from the computer vision domain, is challenging. This is because the mapping from the raw discrete input space (binary programs, source code, network traffic, etc.) to the extracted (or embedded) feature space is typically not invertible nor differentiable. Because of this, when an attacker identifies the ideal perturbations to cause a misclassification, it is usually not obvious how to apply them to the original input—or whether such a transformation is possible at all.

In this chapter we devise state-of-the-art approaches for automatically generating functioning malware that evade machine learning-based detection systems. We provide a formalization to unify vital aspects of problem-space attacks with feature-space attacks and identify key requirements for successful end-to-end attacks in typical security domains. Additionally, we demonstrate the utility of our formalization by using it to identify weaknesses in prior Android evasion attacks and propose a novel attack based on automated software transplantation. We show how such attacks can evade a hardened variant of a state-of-the-art classifier, and be used to facilitate scalable, low-cost universal adversarial perturbations.

4.1 Key Insights

For reference, this chapter provides the following contributions:

- We propose a novel formalization of problem-space attacks (Section 4.3) which lays the foundation for identifying key requirements and commonalities of different domains, proves necessary

and sufficient conditions for problem-space attacks, and allows for the comparison of prior approaches—where existing strategies for adversarial malware generation are among the weakest in terms of attack robustness. We introduce the concept of *side-effect features*, which reveals connections between feature space and problem space, and enables principled reasoning about search strategies for problem-space attacks.

- We propose a novel problem-space attack in the Android malware domain, which relies on automated software transplantation [28] and overcomes limitations of prior work in terms of semantics and preprocessing robustness (Section 4.4). We experimentally demonstrate (Section 4.5) on a dataset of 170K apps from 2017–2018 that it is feasible for an attacker to evade a state-of-the-art malware classifier, DREBIN [19], and its hardened version, Sec-SVM [77]. The time required to generate an adversarial example is in the order of minutes, demonstrating that the "adversarial-malware as a service" scenario is a realistic threat, and existing defenses are not sufficient.
- We demonstrate how our set of problem-space transformations can facilitate attacks using fully realizable *Universal Adversarial Perturbations* (UAPs) which are low-cost perturbations that can be reused across multiple malware examples (Section 4.6).
- We show how introducing problem-space knowledge into robust training procedures can mitigate the threat of realizable attacks. Our defense raises the cost for attackers and disincentivizes the use of powerful UAPs (Section 4.6.4).

The content of this chapter has been previously presented in the following publications:

- Pierazzi F.*, Pendlebury F.*, Cortellazzi J., Cavallaro L. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proc. of the IEEE Symposium of Security and Privacy (S&P)*. 2020.
- Labaca-Castro R., Muñoz-Gonzàlez L., Pendlebury F., Rodosek G. D., Pierazzi F., Cavallaro L. Universal Adversarial Perturbations for Malware. In *arXiv CoRR repository (preprint)*. 2021.

4.2 Overview

Crafting realizable evasive inputs is difficult due to the inverse-feature mapping problem.

Adversarial machine learning (ML) attacks are being studied extensively in multiple domains [34] and pose a major threat to the large-scale deployment of machine learning solutions in security-critical contexts. However, when it comes to generating realizable test-time evasion attacks in the so-called *problem space*, where real input-space objects must be modified to correspond to adversarial feature vectors, there exists a major challenge due to the *inverse feature-mapping problem* [186, 185, 231, 131, 37, 38].

This problem describes how it is often not possible to convert a feature vector into a problem-space object because the feature-mapping function is neither invertible nor differentiable. In addition, the modified problem-space object needs to be a valid, inconspicuous member of the considered domain, and robust to non-ML preprocessing. Existing work has investigated problem-space attacks for various data modalities such as text [10, 169], malicious PDFs [185, 184, 37, 318, 164, 73], Android malware [77, 321], Windows malware [154, 240], network traffic [100, 16, 17, 65], industrial control systems [331], source code attribution [231], malicious Javascript [92], and eyeglass frames to thwart facial recognition [255]. However, while there is a good understanding on how to perform attacks in the feature space (manipulating only the numerical representation of the examples) [51], it is less clear what the requirements are for an attack in the problem space, and how to compare strengths and weaknesses of existing solutions in a principled way.

We propose a novel formalization to unify feature-space and realizable problem-space attacks...

Motivated by examples on software, we propose a novel formalization of problem-space attacks, which lays the foundation for identifying key requirements and commonalities among different domains. We identify four major categories of constraints to be defined at design time: which problem-space *transformations* are available to be performed automatically while looking for an adversarial variant; which object *semantics* must be preserved between the original and its adversarial variant; which non-ML *preprocessing* the attack should be robust to (*e.g.*, image compression, code pruning); and how to ensure that the generated object is a *plausible* member of the input distribution, especially upon manual inspection. We introduce the concept of *side-effect features* as the by-product of trying to generate a problem-space transformation that perturbs the feature space in a certain direction. This allows us to shed light on the relationships between feature space and problem space: we define and prove necessary and sufficient conditions for the existence of problem-space attacks, and identify two main types of search strategies (gradient-driven and problem-driven) for generating problem-space adversarial objects.

...and explore the requirements, constraints, and by-products of problem-space attacks...

We further use our formalization to describe several interesting attacks proposed in both problem space and feature space. This analysis shows that prior promising problem-space attacks in the malware domain [240, 321, 119] suffer from limitations, especially in terms of semantics and preprocessing robustness. For example, Grosse et al. [119] only add individual features to the Android manifest, which preserves semantics, but can be removed with preprocessing (*e.g.*, by detecting unused permissions); moreover, they are constrained by a maximum feature-space perturbation, which we show is less relevant for problem-space attacks. Rosenberg et al. [240] leave artifacts during the app transformation which are easily detected through lightweight non-ML techniques. Yang et al. [321] may significantly alter the semantics of the program (which may

account for the high failure rate observed in their mutated apps), and do not specify which preprocessing techniques they consider. These inspire us to propose, guided by our formalization, a novel problem-space attack in the Android malware domain that overcomes the limitations of existing solutions.

...which allows us to identify and improve upon weaknesses in prior problem-space Android malware evasion attacks.

4.3 Problem-Space Adversarial ML Attacks

We focus on *evasion attacks* [37, 51, 131], where the adversary modifies objects at test time to induce targeted misclassifications. We provide background from related literature on *feature-space* attacks (Section 4.3.1), and then introduce a novel formalization of *problem-space* attacks (Section 4.3.2). Finally, we highlight the main parameters of our formalization by instantiating it on both traditional feature-space and more recent problem-space attacks from related works in several domains (Section 4.3.3). To ease readability, a full list of symbols used is reported in Table 1.

4.3.1 *Feature-Space Attacks*

We remark that all definitions of feature-space attacks (Section 4.3.1) have already been consolidated in related work [34, 51, 77, 119, 279, 131, 72, 178]; we report them for completeness and as a basis for identifying relationships between feature-space and problem-space attacks in the following subsections.

We consider a *problem space* $\mathcal{Z}$ (also referred to as *input space*) that contains objects of a considered domain (*e.g.*, images [51], audio [49], programs [231], PDFs [184]). We assume that each object $z \in \mathcal{Z}$ is associated with a ground-truth label $y \in \mathcal{Y}$, where $\mathcal{Y}$ is the space of possible labels. Machine learning algorithms mostly work on numerical vector data [39], hence the objects in $\mathcal{Z}$ must be transformed into a suitable format for ML processing.

Given a particular model and input domain...

Definition 6 (Feature Mapping) *A feature mapping is a function $\varphi : \mathcal{Z} \longrightarrow \mathcal{X} \subseteq \mathbb{R}^n$ that, given a problem-space object $z \in \mathcal{Z}$, generates an n-dimensional feature vector $\boldsymbol{x} \in \mathcal{X}$, such that $\varphi(z) = \boldsymbol{x}$. This also includes* implicit/latent *mappings, where the features are not observable in the input but are instead implicitly computed by the model (e.g., deep learning [111]).*

Definition 7 (Discriminant Function) *Given an m-class machine learning classifier $g : \mathcal{X} \longrightarrow \mathcal{Y}$, a* discriminant function *$h : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ outputs a real number $h(\boldsymbol{x}, i)$, for which we use the shorthand $h_i(\boldsymbol{x})$, that represents the fitness of object $\boldsymbol{x}$ to class $i \in \mathcal{Y}$. Higher outputs of the discriminant function h_i represent better fitness to class i. In particular, the predicted label of an object $\boldsymbol{x}$ is $g(\boldsymbol{x}) = \hat{y} = \arg\max_{i \in \mathcal{Y}} h_i(\boldsymbol{x})$.*

The purpose of a *targeted* feature-space attack is to modify an object $\boldsymbol{x} \in \mathcal{X}$ with assigned label $y \in \mathcal{Y}$ to an object $\boldsymbol{x}'$ that is

classified to a target class $t \in \mathcal{Y}$, $t \neq y$ (*i.e.,* to modify $\boldsymbol{x}$ so that it is misclassified as a target class t). The attacker can identify a perturbation $\boldsymbol{\delta}$ to modify $\boldsymbol{x}$ so that $g(\boldsymbol{x} + \boldsymbol{\delta}) = t$ by optimizing a carefully-crafted *attack objective function*. We refer to the definition of attack objective function in Carlini and Wagner [51] and in Biggio and Roli [34], which takes into account *high-confidence* attacks and multi-class settings.

Definition 8 (Attack Objective Function) *Given an object $\boldsymbol{x} \in \mathcal{X}$ and a target label $t \in \mathcal{Y}$, an* attack objective function *$f : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ is defined as follows:*

$$f(\boldsymbol{x}, t) = \max_{i \neq t} \{h_i(\boldsymbol{x})\} - h_t(\boldsymbol{x}) \,, \tag{4.1}$$

for which we use the shorthand $f_t(\boldsymbol{x})$. Generally, $\boldsymbol{x}$ is classified as a member of t if and only if $f_t(\boldsymbol{x}) < 0$. An adversary can also enforce a desired attack confidence *$\kappa \in \mathbb{R}$ such that the attack is considered successful if and only if $f_t(\boldsymbol{x}) < -\kappa$.*

The intuition is to minimize f_t by modifying $\boldsymbol{x}$ in directions that follow the negative gradient of f_t, *i.e.,* to get $\boldsymbol{x}$ closer to the target class t.

...evasion attacks may find invalid inputs...

In addition to the attack objective function, a considered problem-space domain may also come with constraints on the modification of the feature vectors. For example, in the image domain the value of pixels must be bounded between 0 and 255 [51]; in software, some features in $\boldsymbol{x}$ may only be added but not removed (*e.g.,* API calls [77]).

Definition 9 (Feature-Space Constraints) *We define Ω as the set of* feature-space constraints, *i.e., a set of constraints on the possible feature-space modifications. The set Ω reflects the requirements of realistic problem-space objects. Given an object $\boldsymbol{x} \in \mathcal{X}$, any modification of its feature values can be represented as a* perturbation vector *$\boldsymbol{\delta} \in \mathbb{R}^n$; if $\boldsymbol{\delta}$ satisfies Ω, we borrow notation from model theory [311] and write $\boldsymbol{\delta} \models \Omega$.*

As examples of feature-space constraints, in the image domain [*e.g.,* 51, 34] the perturbation $\boldsymbol{\delta}$ is subject to an upper bound based on L_p norms ($||\boldsymbol{\delta}||_p \leq \delta_{max}$), to preserve similarity to the original object; in software [*e.g.,* 77, 119], only some features of $\boldsymbol{x}$ may be modified, such that $\boldsymbol{\delta}_{lb} \preceq \boldsymbol{\delta} \preceq \boldsymbol{\delta}_{ub}$ (where $\boldsymbol{\delta}_1 \preceq \boldsymbol{\delta}_2$ implies each element of $\boldsymbol{\delta}_1$ is $\leq$ the corresponding i-th element in $\boldsymbol{\delta}_2$).

We can now formalize the traditional feature-space attack as in related work [51, 37, 77, 34, 213].

Definition 10 (Feature-Space Attack) *Given a machine learning classifier g, an object $\boldsymbol{x} \in \mathcal{X}$ with label $y \in \mathcal{Y}$, and a target label $t \in \mathcal{Y}$, $t \neq y$, the adversary aims to identify a perturbation vector $\boldsymbol{\delta} \in \mathbb{R}^n$ such that $g(\boldsymbol{x} + \boldsymbol{\delta}) = t$. The desired perturbation can be achieved by solving the following optimization problem:*

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta} \in \mathbb{R}^n} \quad f_t(\boldsymbol{x} + \boldsymbol{\delta}) \tag{4.2}$$

$$\textit{subject to:} \quad \boldsymbol{\delta} \models \Omega \,. \tag{4.3}$$

A feature-space attack is successful if $f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0$ (or less than $-\kappa$, if a desired attack confidence is enforced).

...which are not well-defined in the feature space...

Without loss of generality, we observe that the feature-space attacks definition can be extended to ensure that the adversarial example is closer to the training data points (*e.g.,* through the tuning of a parameter λ that penalizes adversarial examples generated in low density regions, as in the mimicry attacks of Biggio et al. [37]).

4.3.2 *Problem-Space Attacks*

This section presents a novel formalization of problem-space attacks and introduces insights into the relationship between feature space and problem space.

Inverse Feature-Mapping Problem. The major challenge that complicates (and, in most cases, prevents) the direct applicability of gradient-driven feature-space attacks to find problem-space adversarial examples is the so-called *inverse feature-mapping problem* [186, 185, 231, 131, 37, 38]. As an extension, Quiring et al. [231] discuss the *feature-problem space dilemma*, which highlights the difficulty of moving in both directions: from feature space to problem space, and from problem space to feature space. In most cases, the feature mapping function φ is not bijective, *i.e., not injective* and *not surjective*. This means that given $z \in \mathcal{Z}$ with features $\boldsymbol{x}$, and a feature-space perturbation $\boldsymbol{\delta}^*$, there is no one-to-one mapping that allows going from $\boldsymbol{x} + \boldsymbol{\delta}^*$ to an adversarial problem-space object z'. Nevertheless, there are two additional scenarios. If φ is not invertible but is *differentiable*, then it is possible to backpropagate the gradient of $f_t(\boldsymbol{x})$ from $\mathcal{X}$ to $\mathcal{Z}$ to derive how the input can be changed in order to follow the negative gradient (*e.g.,* to know which input pixels to perturbate to follow the gradient in the deep-learning latent feature space). If φ is not invertible and not differentiable, then the challenge is to find a way to map the adversarial feature vector $\boldsymbol{x}' \in \mathcal{X}$ to an adversarial object $z' \in \mathcal{Z}$, by applying a transformation to z in order to produce z' such that $\varphi(z')$ is "as close as possible" to $\boldsymbol{x}'$; *i.e.,* to follow the gradient towards the transformation that most likely leads to a successful evasion [154]. In problem-space settings such as software, the function φ is typically not invertible and not differentiable, so the search for transforming z to perform the attack cannot be purely gradient-based.

...and do not map to any possible corresponding object in the problem-space.

In this section, we consider the general case in which the feature mapping φ is not differentiable and not invertible (*i.e.,* the most challenging setting), and we refer to this context to formalize problem-space evasion attacks.

First, we define a *problem-space transformation* operator through which we can alter problem-space objects. Due to their generality, we adapt the code transformation definitions from the *compiler engineering* literature [5, 231] to formalize general problem-space transformations.

Definition 11 (Problem-Space Transformation) *A problem-space transformation $T : \mathcal{Z} \longrightarrow \mathcal{Z}$ takes a problem-space object $z \in \mathcal{Z}$ as input and modifies it to $z' \in \mathcal{Z}$. We refer to the following notation: $T(z) = z'$.*

The possible problem-space transformations are either *addition*, *removal*, or *modification* (*i.e.,* combination of addition and removal). In the case of programs, *obfuscation* is a special case of modification.

Definition 12 (Transformation Sequence) *A transformation sequence $\mathbf{T} = T_n \circ T_{n-1} \circ \cdots \circ T_1$ is the subsequent application of problem-space transformations to an object $z \in \mathcal{Z}$.*

A problem-space object is made evasive through a series of transformations...

Intuitively, given a problem-space object $z \in \mathcal{Z}$ with label $y \in \mathcal{Y}$, the purpose of the adversary is to find a transformation sequence $\mathbf{T}$ such that the transformed object $\mathbf{T}(z)$ is classified into any target class t chosen by the adversary ($t \in \mathcal{Y}$, $t \neq y$). One way to achieve such a transformation is to first compute a feature-space perturbation δ^*, and then modify the problem-space object z so that features corresponding to δ^* are carefully altered. However, in the general case where the feature mapping φ is neither invertible nor differentiable, the adversary must perform a search in the problem-space that approximately follows the negative gradient in the feature space. However, this search is not unconstrained, because the adversarial problem-space object $\mathbf{T}(z)$ must be realistic.

Problem-Space Constraints. Given a problem-space object $z \in \mathcal{Z}$, a transformation sequence $\mathbf{T}$ must lead to an object $z' = \mathbf{T}(z)$ that is valid and realistic. To express this formally, we identify four main types of constraints common to any problem-space attack:

1. *Available transformations*, which describe which modifications can be performed in the problem-space by the attacker (*e.g.,* only addition and not removal).

2. *Preserved semantics*, the semantics to be preserved while mutating z to z', with respect to specific feature abstractions which the attacker aims to be resilient against (*e.g.,* in programs, the transformed object may need to produce the same dynamic call traces). Semantics may also be preserved by construction [*e.g.,* 231].

...which are subject to a set of problem-space constraints that we identify.

3. *Plausibility* (or *Inconspicuousness*), which describes which (qualitative) properties must be preserved in mutating z to z', so that z appears realistic upon manual inspection. For example, often an adversarial image must look like a valid image from the training distribution [51]; a program's source code must look manually written and not artificially or inconsistently altered [231]. In the general case, verification of plausibility may be hard to automate and may require human analysis.

4. *Robustness to preprocessing*, which determines which non-ML techniques could disrupt the attack (*e.g.,* filtering in images, dead code removal in programs).

These constraints have been sparsely mentioned in prior literature [37, 231, 318, 34], but have never been identified together as a set for problem-space attacks. When designing a novel problem-space attack, it is fundamental to explicitly define these four types of constraints, to clarify strengths and weaknesses. We believe that this framework captures all nuances of the current state-of-the-art for a thorough evaluation and comparison, but welcome future research that uses this as a foundation to identify new constraints.

We now formally define the constraints. First, similarly to [77, 34], we define the space of available transformations.

Definition 13 (Available Transformations) *We define $\mathcal{T}$ as the space of* available transformations*, which determines which types of automated problem-space transformations T the attacker can perform. In general, it determines if and how the attacker can add, remove, or edit parts of the original object $z \in \mathcal{Z}$ to obtain a new object $z' \in \mathcal{Z}$. We write $\mathbf{T} \in \mathcal{T}$ if a transformation sequence consists of available transformations.*

They are constrained by the attacker's capability to perform certain transformations...

For example, the pixels of an image may be modified only if they remain within the range of integers 0 to 255 [*e.g.*, 51]; in programs, an adversary may only add valid no-op API calls to ensure that modifications preserve functionality [*e.g.*, 240].

Moreover, the attacker needs to ensure that some semantics are preserved during the transformation of z, according to some feature abstractions. Semantic equivalence is known to be generally undecidable [28, 231]; hence, as in Barr et al. [28], we formalize semantic equivalence through *testing*, by borrowing notation from *denotational semantics* [227].

...the need to retain the malicious functionality of the original object...

Definition 14 (Preserved Semantics) *Let us consider two problem-space objects z and $z' = \mathbf{T}(z)$, and a suite of automated tests Y to verify preserved semantics. We define z and z' to be* semantically equivalent *with respect to Y if they satisfy all its tests $\tau \in \mathrm{Y}$, where $\tau : \mathcal{Z} \times \mathcal{Z} \longrightarrow \mathbb{B}$. In particular, we denote semantics equivalence with respect to a test suite Y as follows:*

$$[\![z]\!]^{\tau} = [\![z']\!]^{\tau}, \forall \tau \in \mathrm{Y}, \tag{4.4}$$

where $[\![z]\!]^{\tau}$ denotes the semantics of z induced during test τ.

Informally, Y consists of tests that are aimed at evaluating whether z and z' (or parts of them) lead to the same abstract representations in a certain feature space. In other words, the tests in Y model preserved semantics. For example, in programs a typical test aims to verify that malicious functionality is preserved; this is done through tests where, given a certain test input, the program produces exactly the same output [28]. Additionally, the attacker may want to ensure that an adversarial program (z') leads to the same instruction trace as its benign version (z)—so as not to raise suspicion in feature abstractions derived from dynamic analysis.

Plausibility is more subjective than semantic equivalence, but in many scenarios it is critical that an adversarial object is inconspicuous when manually audited by a human. In order to be plausible, an analyst must believe that the adversarial object is a valid member of the problem-space distribution.

Definition 15 (Plausibility) *We define Π as the set of (typically) manual tests to verify* plausibility. *We say z looks like a valid member of the data distribution to a human being if it satisfies all tests $\pi \in \Pi$, where $\pi : \mathcal{Z} \longrightarrow \mathbb{B}$.*

...to continue appearing like a plausible member of the valid data distribution to an analyst...

Plausibility is often hard to verify automatically; previous work has often relied on user studies with domain experts to judge the plausibility of the generated objects (*e.g.*, program plausibility in [231], realistic eyeglass frames in [255]). Plausibility in software-related domains may also be enforced by construction during the transformation process, *e.g.*, by relying on automated software transplantation [28, 321].

In addition to semantic equivalence and plausibility, adversarial problem-space objects need to ensure they are robust to non-ML automated *preprocessing* techniques that could alter properties on which the adversarial attack depends, compromising the attack.

Definition 16 (Robustness to Preprocessing) *We define Λ as the set of preprocessing operators an object $z' = \mathbf{T}(z)$ should be resilient to. We say z' is robust to preprocessing if $\mathbf{A}(\mathbf{T}(z)) = \mathbf{T}(z)$ for all $\mathbf{A} \in \Lambda$, where $\mathbf{A} : \mathcal{Z} \longrightarrow \mathcal{Z}$ simulates an expected preprocessing.*

Examples of operators in Λ include compression to remove pixel artifacts (in images), filters to remove noise (in audio), and program analyses to remove dead or redundant code (in programs).

...and to be robust to non-ML preprocessing such as static analysis.

Properties affected by preprocessing are often related to *fragile and spurious features* learned by the target classifier. While taking advantage of these may be necessary to demonstrate the weaknesses of the target model, an attacker should be aware that these brittle features are usually the first to change when a model is improved. Given this, a stronger attack is one that does not rely on them.

As a concrete example, in an attack on authorship attribution, Quiring et al. [231] purposefully omit layout features (such as the use of spaces vs. tabs) which are trivial to change. Additionally, Xu et al. [318] discover the presence of font objects as a critical (but erroneously discriminative) feature following their problem-space attack on PDF malware. These are features that are cheap for an attacker to abuse but can be easily removed by the application of some preprocessing. As a defender, investigation of this constraint will help identify features that are weak to adversarial attacks. Note that knowledge of preprocessing can also be exploited by the attacker (*e.g.*, in *scaling attacks* [315]).

We can now define a fundamental set of problem-space constraint elements from the previous definitions.

Definition 17 (Problem-Space Constraints) *We define the* problem-space constraints $\Gamma = \{\mathcal{T}, \mathrm{Y}, \Pi, \Lambda\}$ *as the set of all constraints satisfying* $\mathcal{T}, \mathrm{Y}, \Pi, \Lambda$. *We write* $\mathbf{T}(z) \models \Gamma$ *if a transformation sequence applied to object* $z \in \mathcal{Z}$ *satisfies all the problem-space constraints, and we refer to this as a* valid *transformation sequence. The problem-space constraints* Γ *determine the feature-space constraints* Ω*, and we denote this relationship as* $\Gamma \vdash \Omega$ *(i.e.,* Γ *determines* Ω*); with a slight abuse of notation, we can also write that* $\Omega \subseteq \Gamma$*, because some constraints may be specific to the problem space (e.g., program size similar to that of benign applications) and may not be possible to enforce in the feature space* $\mathcal{X}$.

Satisfying problem-space constraints may induce unwanted side-effect features...

Side-Effect Features. Satisfying the problem-space constraints Γ further complicates the inverse feature mapping, as Γ is a superset of Ω. Moreover, enforcing Γ may require substantially altering an object z to ensure satisfaction of all constraints during mutations. Let us focus on an example in the software domain, where z is a program with features $\boldsymbol{x}$; if we want to transform z to z' such that $\varphi(z') = \boldsymbol{x} + \boldsymbol{\delta}$, we may want to add to z a program o where $\varphi(o) = \boldsymbol{\delta}$. However, the union of z and o may have features different from $\boldsymbol{x} + \boldsymbol{\delta}$, because other consolidation operations are required (*e.g.,* name deduplication, class declarations, resource name normalization)—which cannot be feasibly computed in advance for each possible object in $\mathcal{Z}$. Hence, after modifying z in an attempt to obtain a problem-space object z' with certain features (*e.g.,* close to $\boldsymbol{x} + \boldsymbol{\delta}$), the attacker-modified object may have some additional features that are not related to the intended transformation (*e.g.,* adding an API which maps to a feature in $\boldsymbol{\delta}$), but are required to satisfy all the problem-space constraints in Γ (*e.g.,* inserting valid parameters for the API call, and importing dependencies for its invocation). We call *side-effect features* $\boldsymbol{\eta}$ the features that are altered in $z' = \mathbf{T}(z)$ specifically for the satisfaction of problem-space constraints. We observe that these features do not follow any particular direction of the gradient, and hence they could have both a positive or negative impact on the classification score.

Analogy with Projection. Figure 4.1 presents an analogy between side-effect features $\boldsymbol{\eta}$ and the notion of *projection* in numerical optimization [39], which helps explain the nature and impact of $\boldsymbol{\eta}$ in problem-space attacks. The right half corresponds to higher values of a discriminant function $h(\boldsymbol{x})$ and the left half to lower values. The vertical central curve (where the heatmap value is equal to zero) represents the decision boundary: objects on the left-half are classified as negative (*e.g.,* benign), and objects on the right-half as positive (*e.g.,* malicious). The goal of the adversary is to conduct a *maximum confidence attack* that has an object misclassified as the negative class. The thick solid line represents the *feasible feature space* determined by constraints Ω, and the thin solid line the *feasible problem space* determined by Γ (which corresponds to two unconnected areas). We assume that the initial object $\boldsymbol{x} \in \mathcal{X}$ is always within the feasible problem space. In this example, the attacker first conducts

...as the attack vector is projected back to the set of feasible problem-space inputs.

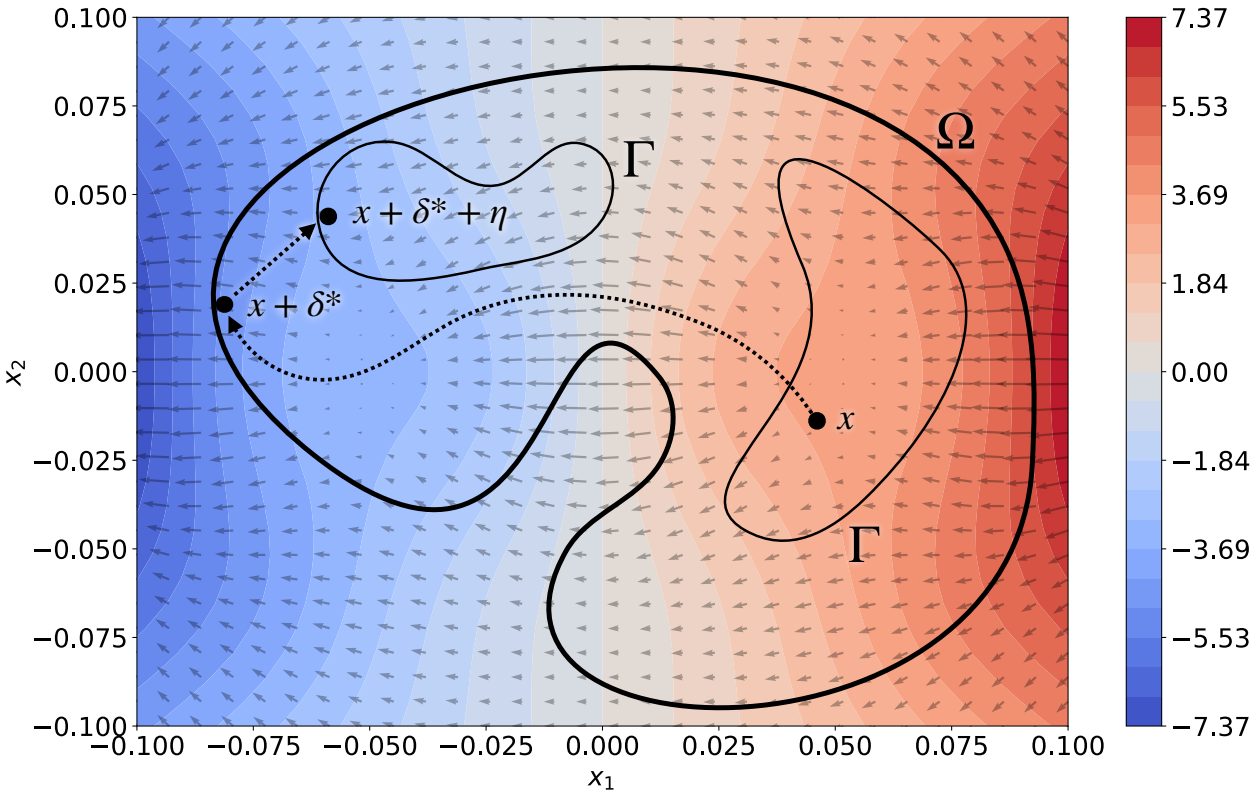


Figure 4.1: Projection of a feature-space attack vector to the *feasible* problem space, resulting in side-effect features $\boldsymbol{\eta}$. Contours depict the value of $h(\boldsymbol{x})$, where negative values indicate the target class of the evasion. Small arrows show directions of the negative gradient. The thick solid line depicts the *feasible* feature space determined by Ω, and the thin solid line that determined by Γ (more restrictive). The dotted arrow depicts the gradient-based attack from $\boldsymbol{x}$ to $\boldsymbol{x} + \boldsymbol{\delta}^*$, which is then projected to $\boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\eta}$ to fit within the feasible problem space.

a gradient-based attack in the feature space on object $\boldsymbol{x}$, which results in a feature vector $\boldsymbol{x} + \boldsymbol{\delta}^*$, which is classified as negative with high-confidence. However, this point is not in the feasibility space of constraints Γ, which is more restrictive than that of Ω. Hence, the attacker needs to find a *projection* that maps $\boldsymbol{x} + \boldsymbol{\delta}^*$ back to the feasible problem-space regions, which leads to the addition of a side-effect feature vector $\boldsymbol{\eta}$.

Definition 18 (Side-Effect Feature Vector) *We define $\boldsymbol{\eta}$ as the* side-effect feature vector *that results from enforcing Γ while choosing a sequence of transformations $\mathbf{T}$ such that $\mathbf{T}(z) \models \Gamma$. In other words, $\boldsymbol{\eta}$ are the features derived from the* projection *of a feature-space attack onto a feasibility region that satisfies problem-space constraints Γ.*

We observe that in settings where the feature mapping φ is neither differentiable nor invertible, and where the problem-space representation is very different from the feature-space representation (*e.g.*, unlike in images or audio), it is generally infeasible or impossible to compute the exact impact of side-effect features on the objective function in advance—because the set of problem-space constraints Γ cannot be expressed analytically in closed-form. Hence the attacker needs to find a transformation sequence $\mathbf{T}$ such that $\varphi(\mathbf{T}(z)) = \varphi(z')$ is within the feasibility region of problem-space constraints Γ.

It is relevant to observe that, in the general case, if an object z_o is added to (or removed from) two different objects z_1 and z_2, it is

possible that the resulting side-effect feature vectors $\boldsymbol{\eta}_1$ and $\boldsymbol{\eta}_2$ are different (*e.g.*, in the software domain [231]).

Considerations on Attack Confidence. There are some important characteristics of the impact of the side-effect features $\boldsymbol{\eta}$ on the attack objective function. If the attacker performs a *maximum-confidence attack* in the feature space under constraints Ω, then the confidence of the problem-space attack will always be *lower or equal* than the one in the feature-space attack. This is intuitively represented in Figure 4.1, where the point is moved to the maximum-confidence attack area within Ω, and the attack confidence is reduced after projection to the feasibility space of the problem space, induced by Γ. In general, the confidence of the feature- and problem-space attacks could be equal, depending on the constraints Ω and Γ, and on the shape of the discriminant function h, which is also not necessarily convex (*e.g.*, in deep learning [111]). In the case of *low-confidence* feature-space attacks, projecting into the problem-space feasibility constraint may result in a positive or negative impact (not known a priori) on the value of the discriminant function. This can be seen from Figure 4.1, where the object $\boldsymbol{x} + \boldsymbol{\delta}^*$ would be found close to the center of the plot, where $h(\boldsymbol{x}) = 0$.

In the majority of cases, side-effect features may have positive or negative impact on the attack success rate.

Problem-Space Attack. We now have all the components required to formalize a problem-space attack.

Definition 19 (Problem-Space Attack) *We define a* problem-space attack *as the problem of finding the sequence of valid transformations* $\mathbf{T}$ *for which the object* $z \in \mathcal{Z}$ *with label* $y \in \mathcal{Y}$ *is misclassified to a target class* $t \in \mathcal{Y}$ *as follows:*

$$\begin{aligned} \textit{argmin}_{\mathbf{T}\in\mathcal{T}} \quad & f_t(\varphi(\mathbf{T}(z))) = f_t(\boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\eta}) && (4.5)\\ \textit{subject to:} \quad & [\![z]\!]^\tau = [\![\mathbf{T}(z)]\!]^\tau, \quad \forall \tau \in \Upsilon && (4.6)\\ & \pi(\mathbf{T}(z)) = 1, \quad \forall \pi \in \Pi && (4.7)\\ & \mathbf{A}(\mathbf{T}(z)) = \mathbf{T}(z), \quad \forall \mathbf{A} \in \Lambda && (4.8) \end{aligned}$$

where $\boldsymbol{\eta}$ *is a side-effect feature vector that separates the feature vector generated by* $\mathbf{T}(z)$ *from the theoretical feature-space attack* $\boldsymbol{x} + \boldsymbol{\delta}^*$ *(under constraints* Ω*). An equivalent, more compact, formulation is as follows:*

$$\begin{aligned} \textit{argmin}_{\mathbf{T}\in\mathcal{T}} \quad & f_t(\varphi(\mathbf{T}(z))) = f_t(\boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\eta}) && (4.9)\\ \textit{subject to:} \quad & \mathbf{T}(z) \models \Gamma\,. && (4.10) \end{aligned}$$

Search Strategy. The typical search strategy for adversarial perturbations in feature-space attacks is based on following the negative gradient of the objective function through some numerical optimization algorithm, such as stochastic gradient descent [34, 51, 49]. However, it is not possible to directly apply gradient descent in the general case of problem-space attacks, when the feature space is not invertible nor differentiable [231, 34]; and it is even more complicated if a transformation sequence $\mathbf{T}$ produces side-effect features $\boldsymbol{\eta} \neq \mathbf{0}$. In the problem space, we identify two main types of search strategy: *problem-driven* and *gradient-driven*. In the problem-driven approach, the search of the optimal $\mathbf{T}$ proceeds heuristically

by beginning with random mutations of the object z, and then learning from experience how to appropriately mutate it further in order to misclassify it to the target class (*e.g.*, using Genetic Programming [318] or variants of Monte Carlo tree search [231]). This approach iteratively uses local approximations of the negative gradient to mutate the objects. The gradient-driven approach attempts to identify mutations that follow the negative gradient by relying on an approximate inverse feature mapping (*e.g.*, in PDF malware [185], in Android malware [321]). If a search strategy equally makes extensive use of both problem-driven and gradient-driven methods, we call it a *hybrid* strategy. We note that search strategies may have different trade-offs in terms of *effectiveness* and *costs*, depending on the time and resources they require. While there are some promising avenues in this challenging but important line of research [158], it warrants further investigation in future work.

Viable attacks can be found through mutating objects randomly or with gradient information as guidance...

Feature-space attacks can still give us some useful information: before searching for a problem-space attack, we can verify whether a feature-space attack exists, which is a necessary condition for realizing the problem-space attack.

Theorem 1 (Necessary Condition for Problem-Space Attacks) *Given a problem-space object $z \in \mathcal{Z}$ of class $y \in \mathcal{Y}$, with features $\varphi(z) = \boldsymbol{x}$, and a target class $t \in \mathcal{Y}$, $t \neq y$, there exists a transformation sequence* $\mathbf{T}$ *that causes* $\mathbf{T}(z)$ *to be misclassified as* t only if *there is a solution for the feature-space attack under constraints Ω. More formally, only if:*

$$\exists \boldsymbol{\delta}^* = \arg \min_{\boldsymbol{\delta} \in \mathbb{R}^n : \boldsymbol{\delta} \models \Omega} f_t(\boldsymbol{x} + \boldsymbol{\delta}) : f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0 \,. \tag{4.11}$$

Proof of Theorem 1. We proceed with a proof by contradiction. Let us consider a problem-space object $z \in \mathcal{Z}$ with features $\boldsymbol{x} \in \mathcal{X}$, which we want to misclassify as a target class $t \in \mathcal{Y}$. Without loss of generality, we consider a low-confidence attack, with desired attack confidence $\kappa = 0$ (see Equation 8). We assume by contradiction that there is no solution to the feature-space attack; more formally, that there is no solution $\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta} \in \mathbb{R}^n : \boldsymbol{\delta} \models \Omega} f_t(\boldsymbol{x} + \boldsymbol{\delta})$ that satisfies $f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0$. We now try to find a transformation sequence $\mathbf{T}$ such that $f_t(\varphi(\mathbf{T}(z))) < 0$. Let us assume that $\mathbf{T}^*$ is a transformation sequence that corresponds to a successful problem-space attack. By definition, $\mathbf{T}^*$ is composed by individual transformations: a first transformation T_1, such that $\varphi(T_1(z)) = \boldsymbol{x} + \boldsymbol{\delta}_1$; a second transformation T_2 such that $\varphi(T_2(T_1(z)) = \boldsymbol{x} + \boldsymbol{\delta}_1 + \boldsymbol{\delta}_2$; a k-th transformation $\varphi(T_k(\cdots T_2(T_1(z)))) = \boldsymbol{x} + \sum_k \boldsymbol{\delta}_k$. We recall that the feature-space constraints are determined by the problem-space constraints, i.e., $\Gamma \vdash \Omega$, and that, with slight abuse of notation, we can write that $\Omega \subseteq \Gamma$; this means that the search space allowed by Γ is smaller or equal than that allowed by Ω. Let us now replace $\sum_k \boldsymbol{\delta}_k$ with $\boldsymbol{\delta}^\dagger$, which is a feature-space perturbation corresponding to the problem-space transformation sequence $\mathbf{T}$, such that $f_t(\boldsymbol{x} + \boldsymbol{\delta}^\dagger) < 0$ (i.e., the sample is misclassified). However, since the constraints imposed by Γ are stricter or equal than those imposed

...but they can always be found if a valid feature-space attack exists...

by Ω, this means that $\delta^\dagger$ must be a solution to $\arg\min_{\delta\in\Omega} f_t(\boldsymbol{x}+\delta)$ such that $f_t(\boldsymbol{x}+\delta^\dagger) < 0$. However, this is impossible, because we hypothesized that there was no solution for the feature-space attack under the constraints Ω. Hence, having a solution in the feature-space attack is a *necessary condition* for finding a solution for the problem-space attack. ■

We observe that Theorem 1 is necessary but *not sufficient* because, although it is not required to be invertible or differentiable, some sort of "mapping" between problem- and feature-space perturbations needs to be known by the attacker. A *sufficient condition* for a problem-space attack, reflecting the attacker's ideal scenario, is knowledge of a set of problem-space transformations which can alter feature values arbitrarily. This describes the scenario for some domains, such as images [51, 112], in which the attacker can modify any pixel value of an image independently.

...and the attacker knows problem-space transformations that can modify any feature by any value...

Theorem 2 (Sufficient Condition for Problem-Space Attacks) *Given a problem-space object $z \in \mathcal{Z}$ of class $y \in \mathcal{Y}$, with features $\varphi(z) = \boldsymbol{x}$, and a target class $t \in \mathcal{Y}$, $t \neq y$, there exists a transformation sequence $\mathbf{T}$ that causes $\boldsymbol{x}$ to be misclassified as t* if *Equation 4.11 and Equation 4.12 are satisfied:*

$$\exists\delta^* = \arg\min_{\delta\in\mathbb{R}^n:\delta\models\Omega} f_t(\boldsymbol{x}+\delta) : f_t(\boldsymbol{x}+\delta^*) < 0 \tag{4.11}$$

$$\forall\delta\in\mathbb{R}^n : \delta\models\Omega, \quad \exists\mathbf{T} : \mathbf{T}(z)\models\Gamma, \varphi(\mathbf{T}(z)) = \boldsymbol{x}+\delta \tag{4.12}$$

Informally, an attacker is always able to find a problem-space attack if a feature-space attack exists (necessary condition) and they know problem-space transformations that can modify any feature by any value (sufficient condition).

Proof of Theorem 2. The existence of a feature-space attack (Equation 4.11) is the necessary condition, which has been proved for Theorem 1. Here we need to prove that, with Equation 4.12, the condition is sufficient for the attacker to find a problem-space transformation that misclassifies the object. Another way to write Equation 4.12 is to consider that the attacker knows transformations that affect individual features only (modifying more than one feature will result as a composition of such transformations). Formally, for any object $z \in \mathcal{Z}$ with features $\varphi(z) = \boldsymbol{x} \in \mathcal{X}$, for any feature-space dimension X_i of $\mathcal{X}$, and for any value $v \in \mathit{domain}(X_i)$, let us assume the attacker knows a valid problem-space transformation sequence $\mathbf{T} : \mathbf{T}(z)\models\Gamma, \varphi(\mathbf{T}(z)) = \boldsymbol{x}'$, such that:

$$x'_i = x_i + v, \qquad x_i \in \boldsymbol{x}, x'_i \in \boldsymbol{x}' \tag{4.13}$$

$$x'_j = x_j, \qquad \forall j \neq i, x_j \in \boldsymbol{x}, x'_j \in \boldsymbol{x}' \tag{4.14}$$

Intuitively, these two equations refer to the existence of a problem-space transformation $\mathbf{T}$ that affects only one feature X_i in $\mathcal{X}$ by any amount $v \in \mathit{domain}(X_i)$. In this way, given *any* adversarial feature-space perturbation δ^*, the attacker is sure to find a transformation

sequence that modifies each individual feature step-by-step. In particular, let us consider $idx_0, \ldots, idx_{q-1}$ corresponding to the $q > 0$ values in δ^* that are different from 0 (i.e., values corresponding to an actual feature-space perturbation). Then, a transformation sequence $\mathbf{T} : \mathbf{T}(z) \models \Gamma, \mathbf{T} = \mathbf{T}^{idx_q-1} \circ \mathbf{T}^{idx_q-2} \circ \cdots \circ \mathbf{T}^{idx_0}$ can always be constructed by the attacker to satisfy $\varphi(\mathbf{T}(z)) = \boldsymbol{x} + \delta^*$. We highlight that we do not consider the existence of a specific transformation in $\mathcal{Z}$ that maps to $\boldsymbol{x} + \delta^*$ because that may not be known by the attacker; hence, the attacker may never learn such a specific transformation. Thus, Equation 4.12 must be valid for all possible perturbations within the considered feature space. ■

In the general case, while there may exist an optimal feature-space perturbation δ^*, there may *not* exist a problem-space transformation sequence $\mathbf{T}$ that alters the feature space of $\mathbf{T}(z)$ exactly so that $\varphi(\mathbf{T}(z)) = \boldsymbol{x} + \delta^*$. This is because, in practice, given a target feature-space perturbation δ^*, a problem-space transformation may generate a vector $\varphi(\mathbf{T}(z)) = \boldsymbol{x} + \delta^* + \boldsymbol{\eta}^*$, where $\boldsymbol{\eta}^* \neq \mathbf{0}$ (*i.e.*, where there may exist at least one i for which $\eta_i \neq 0$) due to the requirement that problem-space constraints Γ must be satisfied. This prevents easily finding a problem-space transformation that follows the negative gradient. Given this, the attacker is forced to apply some search strategy based on the available transformations.

...but realistically their transformations are likely to be constrained and introduce side-effect features.

Corollary 2.1 *If Theorem 2 is satisfied only on a subset of feature dimensions X_i in $\mathcal{X}$, which collectively create a subspace $\mathcal{X}_{eq} \subset \mathcal{X}$, then the attacker can restrict the search space to $\mathcal{X}_{eq}$, for which they know that an equivalent problem/feature-space manipulation exists.*

4.3.3 *Describing problem-space attacks in different domains*

Table 4.1 illustrates the main parameters that need to be explicitly defined while designing problem-space attacks by considering a representative set of adversarial attacks in different domains: images [51], facial recognition [255], text [224], PDFs [318], Javascript [92], code attribution [231], and three problem-space attacks applicable to Android: two from the literature [240, 321] and ours proposed in Section 4.4.

Our formalization can describe prior attacks across a wide variety of domains.

This table shows the expressiveness of our formalization, and how it is able to reveal strengths and weaknesses of different proposals. In particular, we identify some major limitations in two recent problem-space attacks [240, 321]. Rosenberg et al. [240] leave artifacts during the app transformation which are easily detected without the use of machine learning (see Section 4.8 for details), and relies on no-op APIs which could be removed through dynamic analysis. Yang et al. [321] do not specify which preprocessing they are robust against, and their approach may significantly alter the semantics of the program—which may account for the high failure rate they observe in the mutated apps. This inspired us to propose a novel attack that overcomes such limitations.

Table 4.1: Problem-space evasion attacks from prior work across different settings and domains, modeled with our formalization.

		Domains										
		Image Classification [51]	**Facial Recognition** [255]	**Audio** [49]	**Text** [169]	**Code Attribution** [231]	**Javascript** [92]	**PDF** [318]	**Windows** [154]	**Windows RNN** [240]	**Android Transplantation** [321]	**Our Android Attack** (Section 4.4)
Threat Model	Knowledge θ	PK.	PK.	PK.	PK and ZK.	ZK.	ZK.	ZK.	PK.	ZK.	ZK.	PK.
	Feature mapping φ	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.
	Feature space $\mathcal{X}$	Latent feature space of pixels.	Latent feature space of pixels.	Latent feature space of audio stream.	Latent feature space of word embeddings.	Syntactic and lexical static features.	Static syntactic, based on AST, PDG, CFG.	Static (metadata, object keywords and properties, structural).	Feature mapping of MalConv [232].	Dynamic API sequences, static printable strings (also in latent feature space).	Static analysis (RTLD model [321]).	Lightweight static analysis (binary features).
	Problem space $\mathcal{Z}$	Image (pixels).	Printed image (pixels).	Audio (signal).	Text.	Software (source code).	Software (source code).	PDF.	Software (binary).	Software (bytecode).	Software (bytecode).	Software (bytecode).
	Classifier g	Deep learning.	Deep learning.	Deep learning.	LR, CNN, LSTM (PK) and numerous major cloud services (ZK).	Any classifier.	Any classifier.	SVM-RBF (Hidost [271]), RF (PDFRate [264]).	Deep learning (MalConv [232]).	RNN/LSTM variants, and transferability to traditional classifiers (e.g., RF, SVM).	kNN, DT, SVM (and VirusTotal [115]).	Linear SVM (DREBIN [19]) and its hardened version (Sec-SVM [77]).
Problem-Space Constraints	Available Transformations $\mathcal{T}$	(i) Modification of pixel values ($x+\delta \in [0,1]^n$). (ii) Pixel values must be integers from 0 to 255 (*discretization problem*).	(i) Modification of pixel values ($x+\delta \in [0,1]^n$). (ii) Pixel values must be integers from 0 to 255. (iii) Pixels are printable. (iv) Robust to 3D rotations.	(i) Addition of audio noise. (ii) Audio values bounded (i.e., $x+\delta \in [-M,+M]$).	(i) Character-level perturbations. (ii) Word-level perturbations.	(i) Pre-defined set of semantics-preserving code transformations (i.e., modifications). (ii) No changes to the layout of the code.	Transplantation of semantically-equivalent benign ASTs.	Addition/Removal of elements in the PDF tree structure.	Addition of carefully-crafted bytes at the end of the binary.	(i) Addition of no-op API calls with valid parameters. (ii) Repacking of the input malware.	Code addition and modification (within the same program) through *automated software transplantation*.	Code addition through *automated software transplantation*.
	Preserved Semantics Υ	An image should not trivially become an image of another class, so perturbation is constrained	Human subjects retain their original identity and their recognizability to other humans (compared to using full face masks, disguises, etc).	Semantics of original audio preserved by constraining the perturbation	Sentence meaning preserved by (i) replacing like characters (ii) using the GloVe model [224] to swap semantically (not syntactically) similar words.	Source code semantics preserved by construction through use of semantics-preserving transformations.	Malicious semantics preserved by construction through use of AST-based transplantation.	Malicious network functionality is still present (verification with Cuckoo Sandbox).	Malicious code is unaffected by only appending redundant bytes.	API sequences and function return values are unchanged (verification with Cuckoo Monitor).	Malicious semantics preserved, tested by installing and executing each application.	Malicious semantics preserved by construction with *opaque predicates* (newly inserted code is not executed at runtime).
	Robustness to Preprocessing Λ	None explicitly considered.	Discussed but not robust to: the use of specific illumination or distance of the camera.	Robust to: (i) Addition of pointwise random noise (ii) MP3 compression. Discussed but not robust to: Over-the-air playing.	Not explicitly considered.	Robust to: removal of layout features (i.e., use of tabs vs spaces) which are trivial to alter.	Robust to: removal of name inconsistencies of functions and variables.	Discussed but not robust to: removal of spurious features such as presence or absence of font objects (discovered post-attack).	Discussed but not robust to: removal of redundant (non-*text*) bytes.	Robust to: removal of redundant code, undeclared variables, unlinked resources, undefined references, name conflicts.	Not explicitly considered.	Robust to: removal of redundant code, undeclared variables, unlinked resources, undefined references, name conflicts, no-op instructions.
	Plausibility Π	Perturbation constrained ($\|\|\delta\|\|_p \leq \delta_{max}$), to ensure the changes are imperceptible to a human.	(i) Perturbation constrained ($\|\|\delta\|\|_p \leq \delta_{max}$), (ii) Smooth pixel transitions so the eyeglass frames look legitimate with plausible deniability.	Perturbation constrained ($dB_x(\delta) \leq dB_{max}$), so that added noise resembles white background noise largely imperceptible to a human.	(i) Ensure short distance (e.g., edit distance) of modifications (ii) User study to verify plausibility.	The code does not look suspicious and seems written by a human (survey with developers).	By construction through automated AST transplantation (although plausibility is inhibited if certain objects are used, e.g., obsolete ActiveX components).	PDFs can still be parsed and opened by a reader.	None explicitly considered.	The added no-op API calls do not raise errors.	Code is realistic by construction through automated software transplantation.	(i) Code is realistic by construction through use of automated software transplantation. (ii) Mutated apps install and start on an emulator.
Other	Search Strategy	**Gradient-driven.** Stochastic Gradient Descent in the feature space.	**Gradient-driven.** Stochastic Gradient Descent in the feature space.	**Gradient-driven.** Adam optimizer [152] with learning rate 10 and 5,000 max iterations.	**Hybrid** (PK). Gradients used to choose 'top' words. **Problem-driven** (ZK). Without gradients, importance of words is estimated by scoring without each word.	**Problem-driven.** New Monte-Carlo Search algorithm, applied to the problem space.	**Problem-driven.** Search of isomorphic sub-AST graphs in benign samples that are equivalent to malicious sub-ASTs.	**Problem-driven.** Genetic Programming.	**Gradient-driven.** Although the feature mapping is not invertible and not differentiable, the authors devise an algorithm to project byte padding on to the negative gradient.	**Hybrid.** Greedy algorithm selects API calls in order to minimize difference between current and previous iterations w.r.t. the direction of the Jacobian.	**Gradient-driven.** Prioritizing mutations that affect features typical of malware evolution (e.g., phylogenetic trees) and those present in both malware and goodware.	**Gradient-driven.** We use an approximate inverse of the feature mapping, and then a greedy algorithm in the problem space to follow the negative gradient.
	Side-effect features $\boldsymbol{\eta}$	$\boldsymbol{\eta} = \mathbf{0}$	$\boldsymbol{\eta} = \mathbf{0}$	$\boldsymbol{\eta} = \mathbf{0}$	$\boldsymbol{\eta} = \mathbf{0}$	$\boldsymbol{\eta} \simeq \mathbf{0}$	$\boldsymbol{\eta} \neq \mathbf{0}$	$\boldsymbol{\eta} \simeq \mathbf{0}$	$\boldsymbol{\eta} = \mathbf{0}$	$\boldsymbol{\eta} \simeq \mathbf{0}$	$\boldsymbol{\eta} \neq \mathbf{0}$	$\boldsymbol{\eta} \neq \mathbf{0}$

4.4 Attack on Android

Our formalization of problem-space attacks has allowed for the identification of weaknesses in prior approaches to malware evasion applicable to Android [321, 240]. Hence, we propose—through our formalization—a novel problem-space attack in this domain that overcomes these limitations, especially in terms of preserved semantics and preprocessing robustness (see Section 4.3.3 and Section 4.8 for a detailed comparison).

4.4.1 *Threat Model*

The threat model must be defined in terms of attacker *knowledge* and *capability*, as in related literature [34, 277, 51]. While the attacker knowledge is represented in the same way as in the traditional feature-space attacks, their capability also includes the problem-space constraints Γ. For further context, please refer to the discussion on threat models in Section 2.3.

We propose a novel problem-space attack against Android malware detectors...

In our experiments here, we assume an attacker with *perfect knowledge* $\theta_{PK} = (\mathcal{D}, \mathcal{X}, g, \boldsymbol{w})$. This follows Kerckhoffs' principle [147] and ensures a defense does not rely on "security by obscurity" by unreasonably assuming some properties of the defense can be kept secret [52]. Although deep learning has been extensively studied in adversarial attacks, we will show in Chapter 5 that—if retrained frequently—the DREBIN classifier [19] achieves state-of-the-art performance for Android malware detection, which makes it a suitable target classifier for our attack. DREBIN relies on a linear SVM, and embeds apps in a *binary* feature-space $\mathcal{X}$ which captures the presence/absence of components in Android applications in $\mathcal{Z}$ (such as permissions, URLs, Activities, Services, strings). We assume to know classifier g and feature-space $\mathcal{X}$, and train the parameters $\boldsymbol{w}$ with SVM hyperparameter $C = 1$, as in the original DREBIN paper [19]. Using DREBIN also enables us to evaluate the effectiveness of our problem-space attack against a recently proposed hardened variant, Sec-SVM [77]. Sec-SVM enforces more evenly distributed feature weights, which require an attacker to modify more features to evade detection.

We consider an attacker intending to evade detection based on *static analysis*, without relying on code obfuscation as it may increase suspiciousness of the apps [4, 293] (see Section 4.7).

4.4.2 *Available Transformations*

...using automated software transplantation.

We use *automated software transplantation* [28] to extract slices of bytecode (*i.e.*, *gadgets*) from benign *donor* applications and inject them into a malicious *host*, to mimic the appearance of benign apps and induce the learning algorithm to misclassify the malicious host

as benign.[1] An advantage of this process is that we avoid relying on a hardcoded set of transformations [*e.g.*, 231]; this ensures adaptability across different application types and time periods. In this work, we consider only *addition* of bytecode to the malware—which ensures that we do not hinder the malicious functionality.

[1] Our approach is generic and it would be immediate to do the opposite, *i.e.*, transplant malicious code into a benign app. However, this would require a dataset with *annotated* lines of malicious code. For this practical reason and for the sake of clarity of this section, we consider only the scenario of adding benign code parts to a malicious app.

Organ Harvesting. In order to augment a malicious host with a given *benign feature* X_i, we must first extract a bytecode gadget ρ corresponding to X_i from some donor app. As we intend to produce realistic examples, we use *program slicing* [308] to extract a functional set of statements that includes a reference to X_i. The final gadget consists of the this target reference (*entry point* L_o), a forward slice (*organ* o), and a backward slice (*vein* v). We first search for L_o, corresponding to an appearance of code corresponding to the desired feature in the donor. Then, to obtain o, we perform a context-insensitive forward traversal over the donor's System Dependency Graph (SDG), starting at the entry point, transitively including all of the functions called by any function whose definition is reached. Finally, we extract v, containing all statements needed to construct the parameters at the entry point. To do this, we compute a backward slice by traversing the SDG in reverse. Note that while there is only one organ, there are usually multiple veins to choose from, but only one is necessary for the transplantation. When traversing the SDG, class definitions that will certainly be already present in the host are excluded (*e.g.*, system packages such as `android` and `java`). For example, for an Activity feature where the variable `intent` references the target Activity of interest, we might extract the invocation `startActivity(intent)` (entry point L_o), the class implementation of the Activity itself along with any referenced classes (organ o), and all statements necessary to construct `intent` with its parameters (vein v). There is a special case for Activities which have no corresponding vein in the bytecode (*e.g.*, a `MainActivity` or an Activity triggered by an intent filter declared in the Manifest); here, we provide an *adapted vein*, a minimal Intent creation and `startActivity()` call adapted from a previously mined benign app that will trigger the Activity. Note that organs with original veins are always prioritized above those without.

A code gadget that will induce benign features is extracted from a donor app...

Organ Implantation. In order to implant some gadget ρ into a host, it is necessary to identify an injection point L_H where v should be inserted. Implantation at L_H should fulfill two criteria: firstly, it should maintain the syntactic validity of the host; secondly, it should be as unnoticeable as possible so as not to contribute to any violation of plausibility. To maximize the probability of fulfilling the first criterion, we restrict L_H to be between two statements of a class definition in a non-system package. For the second criterion, we take a heuristic approach by using *Cyclomatic Complexity* (CC)—a software metric that quantifies the code complexity of components within the host—and choosing L_H such that we maintain existing homogeneity of CC across all components. Finally, the host entry point L_H is inserted into a *randomly chosen* function among

...and inconspicuously implanted into the host app.

those of the selected class, to avoid creating a pattern that might be identified by an analyst.

4.4.3 *Preserved Semantics*

Given an application z and its modified (adversarial) version z', we aim to ensure that z and z' lead to the same dynamic execution, *i.e.,* the malicious behavior of the application is preserved. We enforce this by construction by wrapping the newly injected execution paths in conditional statements that always return `False`. This guarantees the newly inserted code is never executed at runtime—so users will not notice anything odd while using the modified app. In Section 4.4.4, we describe how we generate such conditionals without leaving artifacts.

We ensure the benign code cannot disrupt malicious functionality at runtime...

To further preserve semantics, we also decide to omit `intent-filter` elements as transplantation candidates. For example, an `intent-filter` could declare the app as an eligible option for reading PDF files; consequently, whenever attempting to open a PDF file, the user would be able to choose the host app, which (if selected) would trigger an Activity defined in the transplanted benign bytecode—violating our constraint of preserving dynamic functionality.

4.4.4 *Robustness to Preprocessing*

Program analysis techniques that perform redundant code elimination would remove unreachable code. Our evasion attack relies on features associated with the transplanted code, and to preserve semantics we need conditional statements that always resolve to `False` at runtime; so, we must subvert static analysis techniques that may identify that this code is never executed. We achieve this by relying on *opaque predicates* [202], *i.e.,* carefully constructed obfuscated conditions where the outcome is always known at design time (in our case, `False`), but the actual truth value is difficult or impossible to determine during a static analysis.

...and use opaque predicates to ensure the injected code is not easily detectable by static analysis.

To ensure the intractability of such an analysis, we follow the work of Moser et al. [202] and build opaque predicates using a formulation of the 3-SAT problem such that resolving the truth value of the predicate is equivalent to solving the NP-complete 3-SAT problem.

The k-satisfiability (k-SAT) problem asks whether the variables of a Boolean logic formula can be consistently replaced with `True` or `False` in such a way that the entire formula evaluates to `True`; if so the formula is *satisfiable*. Such a formula is easily expressed in its conjunctive normal form:

$$\bigwedge_{i=1}^{m}(V_{i1} \vee V_{i2} \vee ... \vee V_{ik}),$$

where $V_{ij} \in \{v_1, v_2, ..., v_n\}$ are Boolean variables and k is the number of variables per clause.

Listing 4.1: Example opaque predicate wrapping an *adapted vein* that calls code with benign features. Java is shown for clarity, the actual transplantation occurs with Dalvik bytecode. While ideal Random k-SAT parameters are given here, actual values are synthesized through JSketch with some variance to avoid fingerprints.

```
void opaque() {
  Random random = new Random();
  this();
  boolean[] arrayOfBoolean = new boolean[40];
  byte b1;
  for (b1 = 0; b1 < arrayOfBoolean.length; b1++)
    arrayOfBoolean[b1] = random.nextBoolean();
  b1 = 1;
  for (byte b2 = 0; b2 < 184.0D; b2++) {
    boolean bool = false;
    for (byte b = 0; b < 3; b++)
      bool |= arrayOfBoolean[random.nextInt(arrayOfBoolean.length)];
    if (!bool)
      b1 = 0;
  }
  if (b1 != 0) {
    // Beginning of adapted vein
    Context context = ((Context)this).getApplicationContext();
    Intent intent = new Intent();
    this(this, h.a(this, cxim.qngg.TEhr.sFiQa.class));
    intent.putExtra("1", h.p(this));
    intent.addFlags(268435456);
    startActivity(intent);
    h.x(this);
    return;
    // End of adapted vein
  }
}
```

Importantly, when $k = 3$, formulas are only NP-Hard in the worst case—30% of 3-SAT problems are in P [249]. This baseline guarantee is not sufficient as our injected code should never execute. Additionally, we require a large number of random predicates to reduce commonality between the synthetic portions of our generated examples.

To consistently generate NP-Hard k-SAT problems we use *Random k-SAT* [249] in which there are 3 parameters: the number of variables n, the number of clauses m, and the number of literals per clause k.

Our opaque predicates rely on the hardness of 3-SAT...

To construct a 3-SAT formula, m clauses of length 3 are generated by randomly choosing a set of 3 variables from the n available, and negating each with probability 50%. An empirical study by Selman et al. [249] showed that n should be at least 40 to ensure the formulas are hard to resolve. Additionally, they show that formulas with too few clauses are *under-constrained* while formulas with too many clauses are *over-constrained*, both of which reduce the search time. These experiments led to the following conjecture.

Threshold Conjecture [249]. Let us define c^* as the threshold at which 50% of the formulas are satisfiable. For $m/n < c^*$, as $n \to \infty$, the formula is satisfiable with probability 100%, and for $m/n > c^*$, as $n \to \infty$, the formula is unsatisfiable with probability 100%.

The current state-of-the-art for c^* is $3.42 < c^* \approx 4.3 < 4.51$ for 3-SAT [249, 142, 197]. We use this conjecture to ensure that the

formulas used for predicates are unsatisfiable with high probability, i.e., that the predicate is likely a contradiction and will always evalute to `False`.

...which our tests show will defeat analyses for stripping out dead/redundant code...

Additionally we discard any generated formulas that fall into two special cases of 3-SAT that are polynomially solvable:

- **2-SAT:** The construction may be 2-SAT if it can be expressed as a logically equivalent 2CNF formula [163].
- **Horn-SAT:** If at most one literal in a clause is positive, it is a *Horn clause*. If all clauses are Horn clauses, the formula is Horn-SAT and solvable in linear time [81].

We tested 100M Random 3-SAT trials using the fixed clause-length model with parameters $n \simeq 40, m \simeq 184, c^* \simeq 4.6$. All (100%) of the generated constructions were unsatisfiable (and evaluated to `False` at runtime) which aligns with the findings of Selman et al. [249]. This probability is sufficient to prevent execution with near certainty.

...and can be automatically synthesized using JSketch [137], with some variation to avoid artifacts.

To further reduce artifacts introduced by reusing the same predicate, we use *JSketch* [137], a sketch-based program synthesis tool, to randomly generate new predicates prior to injection with some variation while maintaining the required properties. Post-transplantation, we verify for each adversarial example that Soot's program optimizations have not been able to recognize and eliminate them. An example of a generated opaque predicate (rendered in equivalent Java rather than Dalvik bytecode) is shown in Listing 4.1.

4.4.5 *Plausibility*

In our model, an example is satisfactorily plausible if it resembles a real, functioning Android application (*i.e.*, is a valid member of the problem-space $\mathcal{Z}$). Our methodology aims to maximize the plausibility of each generated object by injecting full slices of bytecode from *real* benign applications. There is only one case in which we inject artificial code: the opaque predicates that guard the entry point of each gadget (see Listing 4.1 for an example). In general, we can conclude that plausibility is guaranteed *by construction* thanks to the use of automated software transplantation [28]. This contrasts with other approaches that inject *standalone* API calls and URLs or *no-op* operations [*e.g.*, 240] that are completely orphaned and unsupported by the rest of the bytecode (*e.g.*, an API call result that is never used).

Smoke tests ensure the app has not been corrupted.

We also practically assess that each mutated app still functions properly after modification by installing and running it on an Android emulator. Although we are unable to thoroughly explore every path of the app in this automated manner, it suffices as a smoke test to ensure that we have not fundamentally damaged the structure of the app.

Algorithm 1: Initialization Phase (Ice-Box Creation)

Input: Discriminant function $h(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$, which classifies $\boldsymbol{x}$ as malware if $h(\boldsymbol{x}) > 0$, otherwise as goodware.
Minimal app $z_{min} \in \mathcal{Z}$ with features $\varphi(z_{min}) = \boldsymbol{x}_{min}$.
Parameters: Number of features to consider n_f; number of donors per-feature n_d.
Output: Ice-box of harvested organs with feature vectors.

ice-box $\leftarrow$ {} Empty key-value dictionary.
$L \leftarrow$ List of pairs (w_i, i), sorted by increasing value of w_i.
$L' \leftarrow$ First n_f elements of L, then remove any entry with $w_i \geq 0$.
for (w_i, i) *in* L' **do**
  ice-box [i] $\leftarrow$ [] Empty list for gadgets with feature i.
  while length(ice-box [i])$< n_d$ **do**
    $z_j \leftarrow$ Randomly sample a benign app with feature $x_i = 1$.
    Extract gadget $\rho_j \in \mathcal{Z}$ with feature $x_i = 1$ from z_j.
    $s \leftarrow$ Software stats of ρ_j
    $z' \leftarrow$ Inject gadget ρ_j in app z_{min}.
    $(\boldsymbol{x}_{min} \vee \boldsymbol{e}_i \vee \boldsymbol{\eta}_j) \leftarrow \varphi(z')$ $\boldsymbol{e}_i$ is a one-hot vector.
    $\boldsymbol{r}_j \leftarrow (\boldsymbol{e}_i \vee \boldsymbol{\eta}_j) \leftarrow \varphi(z') \wedge \neg \boldsymbol{x}_{min}$ Gadget features obtained through set difference.
    if $h(\boldsymbol{r}_j) > 0$ **then**
      Discard the gadget;
    else
      Append $(\rho_j, \boldsymbol{r}_j, s)$ to ice-box [i]. Store gadget
return ice-box;

4.4.6 *Search Strategy*

We propose a *gradient-driven* search strategy based on a *greedy algorithm*, which aims to follow the gradient direction by transplanting a gadget with benign features into the malicious host. There are two main phases: *Initialization* (Ice-Box Creation) and *Attack* (Adversarial Program Generation).

Our attack uses a greedy gradient-driven search strategy...

Algorithm 1 and Algorithm 2 describe in detail the two main phases of our search strategy: organ harvesting and adversarial program generation. For the sake of simplicity, we describe a low-confidence attack, i.e., the attack is considered successful as soon as the classification score is below zero. It is immediate to consider high-confidence variations (as we evaluate in Section 4.5).

Initialization Phase (Ice-Box Creation). We first harvest gadgets from potential donors and collect them in an *ice-box* G, which is used for transplantation at attack time. The main reason for this, instead of looking for gadgets on-the-fly, is to have an immediate estimate of the *side-effect features* when each gadget is considered for transplantation. Looking for gadgets on-the-fly is possible, but may lead to less optimal solutions and uncertain execution times.

...where a repository of gadgets are first extracted from potential donors and added to an 'ice-box'.

For the initialization we aim to gather gadgets that move the score of an object towards the benign class (*i.e.*, negative score), hence we consider the classifier's top n_f benign features (*i.e.*, with negative weight). For each of the top-n_f features, we extract n_d candidate gadgets, excluding those that lead to an overall positive (*i.e.*, malicious) score. We recall that this may happen even for benign features since the context extracted through forward and backward slicing may contain many other features that are indicative of maliciousness. We empirically verify that with $n_f = 500$ and $n_d = 5$ we

are able to create a successfully evasive app for all the malware in our experiments. It is important to observe that the ice-box can be expanded over time, as long as the target classifier does not change its weights significantly. Algorithm 1 reports the detailed steps of the initialization phase.

To estimate the side-effect feature vectors for the gadgets, we inject each into a *minimal app* z_{min}, *i.e.,* an Android app we have developed with minimal functionality. Note that when using z_{min} to calculate the features that will be induced by a gadget, features in the corresponding feature vector $\boldsymbol{x}_{min}$ should be noted and dealt with accordingly (i.e., discounted). In our case x_{min} contained the following three features:

```
{ "intents::android_intent_action_MAIN": 1,
  "intents::android_intent_category_LAUNCHER": 1,
  "activities::_MainActivity": 1 }
```

Attack Phase. We aim to automatically mutate z into z' so that it is misclassified as goodware, *i.e.,* $h(\varphi(z')) < 0$, by transplanting harvested gadgets from the ice-box G. First we search for the list of ice-box gadgets that should be injected into z. Each gadget ρ_j in the ice-box G has feature vector $\boldsymbol{r}_j$ which includes the desired feature and side-effect features. We consider the actual feature-space contribution of gadget i to the malicious host z with features $\boldsymbol{x}$ by performing the set difference of the two binary vectors, $\boldsymbol{r}_j \wedge \neg \boldsymbol{x}$. We then sort the gadgets in order of decreasing negative contribution, which ideally leads to a faster convergence of z's score to a benign value. Next we filter this candidate list to include gadgets *only if* they satisfy some practical feasibility criteria. We define a *check_feasibility* function which implements some heuristics to limit the excessive increase of certain statistics which would raise suspiciousness of the app. Preliminary experiments revealed a tendency to add too many permissions to the Android Manifest, hence, we empirically enforce that candidate gadgets add no more than 1 new permission to the host app. Moreover, we do not allow addition of permissions listed as *dangerous* in the Android documentation [113]. The other app statistics remain reasonably within the distribution of benign apps (more discussion in Section 4.5), and so we decide not to enforce a limit on them. The remaining candidate gadgets are iterated over and for each candidate ρ_j, we combine the gadget feature vector $\boldsymbol{r}_j$ with the input malware feature vector $\boldsymbol{x}$, such that $\boldsymbol{x}' = \boldsymbol{x} \vee \boldsymbol{r}_j$. We repeat this procedure until the updated $\boldsymbol{x}'$ is classified as goodware (for low-confidence attacks) or until an attacker-defined confidence level is achieved (for high-confidence attacks). Finally, we inject all the candidate gadgets at once through automated software transplantation, and check that problem-space constraints are verified and that the app is still classified as goodware. Algorithm 2 reports the detailed steps of the attack phase. For the sake of simplicity, we describe a low-confidence attack, i.e., the attack is considered successful as soon as the classification score

After determining the optimal features to add, gadgets are injected such that the impact of side-effect features is minimized...

...subject to some heuristics to ensure problem-space constraints are not violated.

Algorithm 2: Attack Phase (Adv. Program Generation)

Input: Discriminant function $h(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$, which classifies $\boldsymbol{x}$ as malware if $h(\boldsymbol{x}) > 0$, otherwise as goodware. Malware app $z \in \mathcal{Z}$. Ice-box G.
Parameters: Problem-space constraints.
Output: Adversarial app $z' \in \mathcal{Z}$ such that $h(\varphi(z')) < 0$.

$\mathcal{T} \leftarrow$ Transplantation through gadget addition.
$\Upsilon \leftarrow$ Smoke test through app installation and execution in emulator.
$\Pi \leftarrow$ Plausibility by-design through code consolidation.
$\Lambda \leftarrow$ Artifacts from last column of Table 4.1.
$\Gamma \leftarrow \{\mathcal{T}, \Upsilon, \Pi, \Lambda\}$; $s_z \leftarrow$ Software stats of z; $\boldsymbol{x} \leftarrow \varphi(z)$; $L \leftarrow$ [] Empty list.
$\mathbf{T}(z) \leftarrow$ Empty sequence of problem-space transformations.
for $(\rho_j, \boldsymbol{r}_j, s)$ *in* G **do**
    $\boldsymbol{d}_j \leftarrow \boldsymbol{r}_j \wedge \neg\boldsymbol{x}$ Feature-space contribution of gadget j.
    $score_j \leftarrow h(\boldsymbol{d}_j)$ Impact on decision score.
    Append the pair $(score_j, i, j)$ to L Feature i, Gadget j.
$L' \leftarrow$ Sort L by increasing $score_j$ Negative scores first.
for $(score_j, i, j)$ *in* L' **do**
    if z *has* $x_i = 1$ **then**
        Do nothing; Feature i already present.
    else if z *has* $x_i = 0$ **then**
        $(\rho_j, \boldsymbol{r}_j, s) \leftarrow$ element j in ice-box G
        if check_feasibility(s_z, s) *is True* **then**
            $\boldsymbol{x} \leftarrow (\boldsymbol{x} \vee \boldsymbol{e}_i \vee \boldsymbol{\eta}_j)$ Update features of z.
            Append transplantation $T \in \mathcal{T}$ of gadget ρ_j in $\mathbf{T}(z)$.
            if $h(\boldsymbol{x}) < 0$ **then**
                Exit from cycle; Attack gadgets found.
$z' \leftarrow$ Apply transformation sequence $\mathbf{T}(z)$ Inject chosen gadgets.
if $h(\varphi(z')) < 0$ **and** $\mathbf{T}(z) \models \Gamma$ **then**
    return z'; Attack successful.
return Failure;

is below zero. It is immediate to consider high-confidence variations (as we evaluate in Section 4.5).

4.5 Experimental Evaluation

We evaluate the effectiveness of our novel problem-space Android attack, in terms of success rate and required time—and also when in the presence of feature-space defenses.

4.5.1 *Experimental Settings*

Prototype. We create a prototype of our novel problem-space attack (Section 4.4) using a combination of Python for the ML functionality and Java for the program analysis operations; in particular, to perform transplantations in the problem-space we rely on FlowDroid [21], which is based on Soot [295]. We release the code of our prototype to other academic researchers (see front matter). We ran all experiments on an Ubuntu VM with 48 vCPUs, 290GB of RAM, and NVIDIA Tesla K40 GPU.

Classifiers. As defined in the threat model (Section 4.4.1), we consider the DREBIN classifier [19], based on a binary feature space and a linear SVM, and its recently proposed hardened variant, Sec-

SVM [77], which requires the attacker to modify more features to perform an evasion.

We have access to a working Python implementation of DREBIN based on *sklearn*, *androguard*, and *aapt*, and we rely on *LinearSVC* classifier with C=1 as in Arp et al. [19].

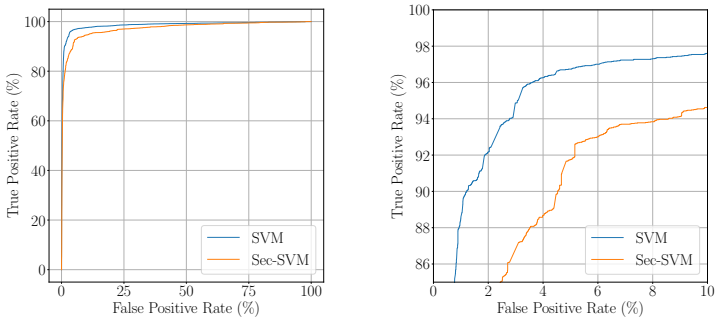
We test our attack against an SVM-based detector, DREBIN [19]...

To have have full control of the training procedure, we approximate the linear SVM as a *single-layer* neural network (NN) using PyTorch [218]. We recall that the main intuition behind Sec-SVM is that classifier weights are distributed more evenly in order to force an attacker to modify more features to evade detection. Hence, we modify the training procedure so that the Sec-SVM weights are bounded by a *maximum weight value* k at each training optimization step. Similarly to Demontis et al. [77], we perform feature selection for computational efficiency, since PyTorch does not support sparse vectors. We use an l_2 (Ridge) regularizer to select the top 10,000 with negligible reduction in AUROC. This performance retention follows from recent results that shows SVM tends to overemphasize a subset of features [191]. To train the Sec-SVM, we perform an extensive hyperparameter grid-search: with Adam [152] and Stochastic Gradient Descent (SGD) optimizers; training epochs of 5 to 100; batch sizes from 2^0 to 2^{12}; learning rate from 10^0 to 10^{-5}. We identify the best single-layer NN configuration for our training data to have the following parameters: SGD, batch size 1024, learning rate 10^{-4}, and 75 training epochs. We then perform a grid-search of the Sec-SVM hyperparameter k (i.e., the maximum weight absolute value [77]) by clipping weights during training iterations. We start from $k = w_{max}$, where $w_{max} = \max_i(w_i)$ for all features i; we then continue reducing k until we reach a weight distribution similar to that reported in [77], while allowing a maximum performance loss of 2% in AUROC. In this way, we identify the best value for our setting as $k = 0.2$.

...and its hardened variant, Sec-SVM [77].

Figure 4.2 reports the AUROC for the DREBIN classifier [19] in SVM and Sec-SVM modes. The SVM mode has been evaluated using the `LinearSVC` class of scikit-learn [221] that utilizes the LIBLINEAR library [91]; as in the DREBIN paper [19], we use hyperparameter C=1. The performance degradation of the Sec-SVM compared to the baseline SVM shown in Figure 4.2 is in part related to the defense itself (as detailed in [77]), and in part due to minor convergence issues (since our single-layer NN converges less effectively than the LIBLINEAR implementation of scikit-learn). We have verified with Demontis et al. [77] the correctness of our Sec-SVM implementation and its performance, for the analysis performed in this work.

Attack Confidence. We consider two attack settings: *low-confidence* (**L**) and *high-confidence* (**H**). The (L) attack merely overcomes the decision boundary (so that $h(\boldsymbol{x}) < 0$). The (H) attack maximizes the distance from the hyperplane into the goodware region; while generally this distance is unconstrained, here we set it to be $\leq$ the negative scores of 25% of the benign apps (*i.e.*, within their interquar-



(a) ROC (b) ROC (Zoom)

Figure 4.2: Performance of SVM and Sec-SVM in absence of adversarial attacks.

tile range). This avoids making superfluous modifications, which may only increase suspiciousness or the chance of transplantation errors, while being closer in nature to past mimicry attacks [37].

Dataset. We collect apps from AndroZoo [8], a large-scale dataset with timestamped Android apps crawled from different stores, and with VirusTotal summary reports. We use the labeling criteria of Miller et al. [194]: an app is considered *goodware* if it has 0 VirusTotal detections, as *malware* if it has 4+ VirusTotal detections, and is discarded as *grayware* if it has between 1 and 3 VirusTotal detections. To ensure spatial bias does not affect the evaluation we use an average of 10% malware (see Section 5.3.3). The final dataset contains ~170K recent Android applications, dated between Jan 2017 and Dec 2018, specifically 152,632 goodware and 17,625 malware.

We use a dataset of 152,632 goodware and 17,625 malware from the Androzoo repository [8]...

Dataset Split. In Chapter 5 (previously published as Pendlebury et al. [223]) we demonstrate that in non-stationary contexts such as Android malware, if time-aware splits are not considered, then the results may be inflated due to *concept drift* (*i.e.*, changes in the data distribution). However, here we aim to specifically evaluate the effectiveness of a single adversarial attack. If we were to perform a time-aware split, it would be impossible to determine whether the success rate of our ML-driven adversarial attack was due to an intrinsic weakness of the classifier or due to wider evolution of malware (*i.e.*, the introduction of new non-ML techniques malware developers rely on to evade detection). Hence, we perform a *random split* of the dataset to simulate an *absence of concept drift* [223]; this also represents the most challenging scenario for an attacker, as they aim to mutate a test object coming from the same distribution as the training dataset (on which the classifier likely has higher confidence). In particular, we consider a 66% training and 34% testing random split.[2]

...with a random training-test split of 2:1 to ensure greater concept drift does not affect the measurement.

Testing. The test set contains a total of 5,952 malware. The statistics reported in the remainder of this section refer only to *true positive* malware (5,330 for SVM and 4,108 for Sec-SVM), *i.e.*, we create adversarial variants only if the app is detected as malware by the classifier under evaluation. Intuitively, it is not necessary to make an adversarial example of a malware application that is already misclassified as goodware; hence, we avoid inflating results by

[2] We consider only one split due to the overall time required to run the experiments. Including some prototype overhead, it requires about one month to run all configurations.

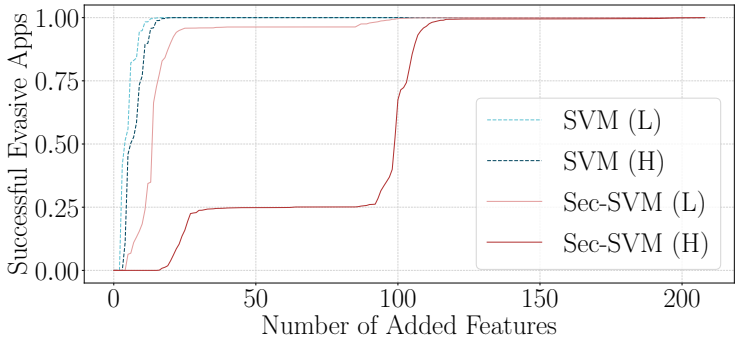


Figure 4.3: Cumulative distribution of features added to adversarial malware (out of a total 10,000 features remaining after feature selection).

removing false negative objects from the dataset. During the transplantation phase of our problem-space attack some errors occur due to bugs and corner-case errors in the FlowDroid framework [21].

We performed extensive troubleshooting of FlowDroid [21] to reduce the number of transplantation failures, and the transplantations without FlowDroid errors in the different configurations are as follows: 89.5% for SVM (L), 85% for SVM (H), 80.4% for Sec-SVM (L), 73.3% for Sec-SVM (H). Some examples of the errors encountered include: inability to output large APKs when the app's SDK version is less than 21; a bug triggered in AXmlWriter, the third party component used by FlowDroid, when modifying app Manifests; and FlowDroid injecting system libraries found on the classpath when they should be excluded.

Our results focus on the ~4–5K true positives (model dependent), omitting failed transformations caused by third-party tooling...

Since these errors are related to implementation limitations of the FlowDroid research prototype, and not conceptual errors, the success rates in the remainder of this section refer only to applications that did not throw FlowDroid exceptions during the transplantation phase.

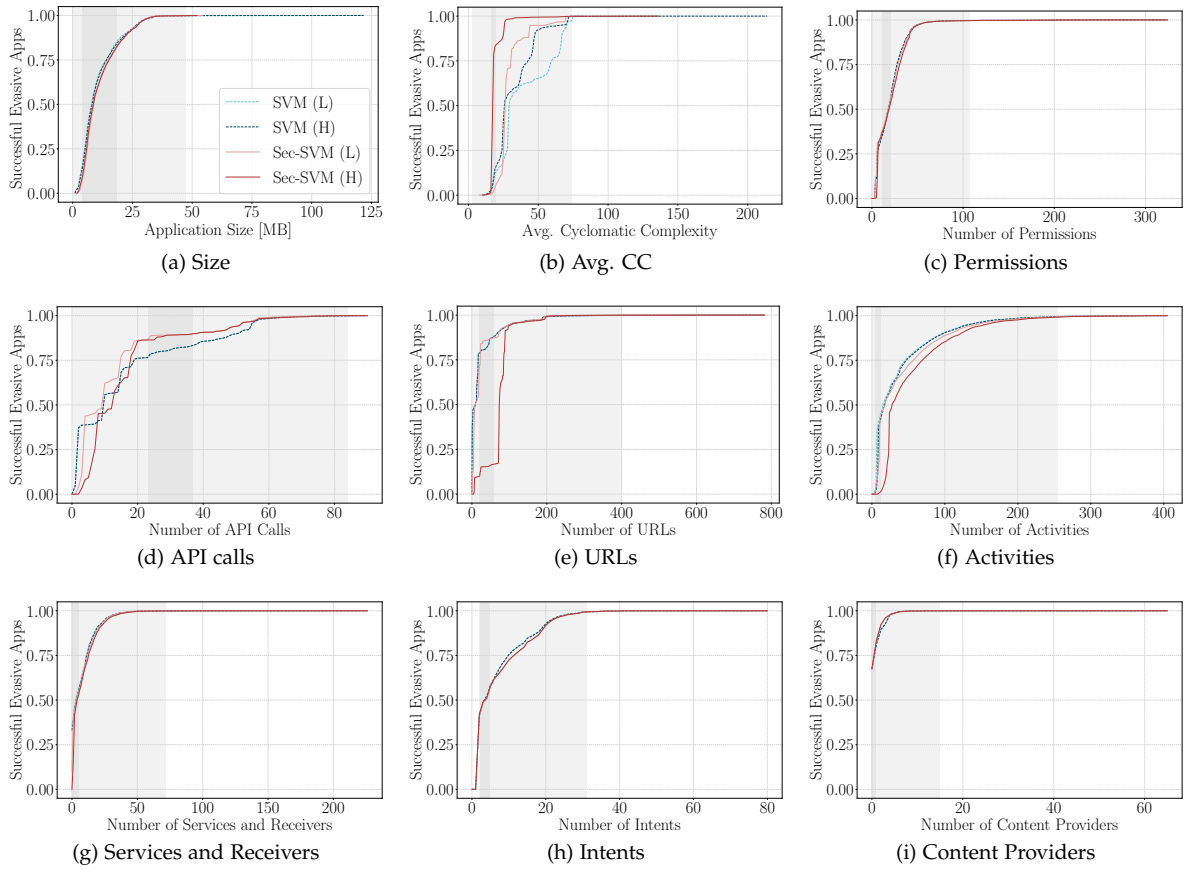
4.5.2 *Evaluation*

We analyze the performance of our Android problem-space attack in terms of runtime cost and successful evasion rate. An attack is successful if an app z, originally classified as malware, is mutated into an app z' that is classified as goodware and satisfies the problem-space constraints.

Figure 4.2 reports the AUROC of SVM and Sec-SVM on the DREBIN feature space in absence of attacks. As expected [77], Sec-SVM sacrifices some detection performance in return for greater feature-space adversarial robustness.

...of which 100% can be made adversarial without violating problem-space constraints...

Attack Success Rate. We perform our attack using *true positive* malware from the test set, *i.e.*, all malware objects correctly classified as malware. We consider four settings depending on the defense algorithm and the attack confidence: SVM (L), SVM (H), Sec-SVM (L), and Sec-SVM (H). In absence of FlowDroid exceptions (see Section 4.5.1), we are able to create an evasive variant for each malware in all four configurations. In other words, we achieve a misclassification rate of 100.0% on the successfully generated apps,



(a) Size (b) Avg. CC (c) Permissions

(d) API calls (e) URLs (f) Activities

(g) Services and Receivers (h) Intents (i) Content Providers

Figure 4.4: Statistics of the evasive malware variants. The dark gray area highlights the interquartile range for benign apps; the light gray area is based on the 3σ rule and highlights the range between 0.15% and 99.85% of the benign distribution, showing the new apps are reasonably plausible.

where the problem-space constraints are satisfied by construction (as defined in Section 4.4). Figure 4.3 reports the cumulative distribution of features added when generating evasive apps for the four different configurations. As expected, Sec-SVM requires the attacker to modify more features, but here we are no longer interested in the feature-space properties, since we are performing a problem-space attack. This demonstrates that measuring attacker effort with L_p perturbations as in the original Sec-SVM evaluation [77] *overestimates* the robustness of the defense and is better assessed using our framework (Section 4.3).

While the plausibility problem-space constraint is satisfied by design by transplanting only realistic existing code, it is informative to analyze how the statistics of the evasive malware relate to the corresponding distributions in benign apps. Figure 4.4 reports the cumulative distribution of app statistics across the four settings: the X-axis reports the statistics values, whereas the Y-axis reports the cumulative percentage of evasive malware apps. We also shade two gray areas: a *dark gray area* between the first quartile q_1 and third quartile q_3 of the statistics for the benign applications; the *light gray area* refers to the 3σ rule and reports the area within the 0.15% and 99.85% of the benign apps distribution.

Figure 4.4 shows that while evading Sec-SVM tends to cause a shift towards the higher percentiles of each statistic, the vast majority of apps fall within the gray regions in all configurations. We

...demonstrating that L_p constraints overestimate robustness in this setting.

note that this is just a qualitative analysis to verify that the statistics of the evasive apps roughly align with those of benign apps; it is not sufficient to have an anomaly in one of these statistics to determine that an app is malicious (otherwise, very trivial rules could be used for malware detection itself, and this is not the case). We also observe that there is little difference between the statistics generated by Sec-SVM and by traditional SVM; this means that greater feature-space perturbations do not necessarily correspond to greater perturbations in the problem-space, reinforcing the feasibility and practicality of evading Sec-SVM.

Our attack is a scalable and practical threat with an average runtime under 2 minutes.

Runtime Overhead. The time to perform the search strategy occurring in the feature space is almost negligible; the most demanding operation is in the actual code modification. Figure 4.5 depicts the distribution of injection times for our test set malware which is the most expensive operation in our approach while the rest is mostly pipeline overhead. The time spent per app is low: in most cases, less than 100 seconds, and always less than 2,000 seconds (~33 mins). The low runtime cost suggests that it is feasible to perform this attack at scale and reinforces the need for new defenses in this domain.

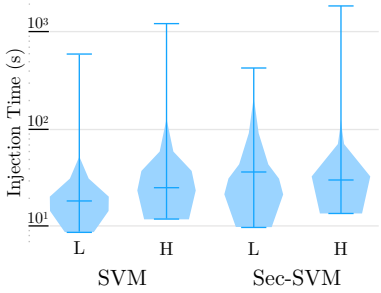


Figure 4.5: Injection times per adversarial app.

4.6 Realizable Universal Adversarial Perturbations

The ability to generate realizable problem-space malware opens up classes of attacks beyond input-specific evasion. Universal Adversarial Perturbations (UAPs) are a class of perturbations where a single perturbation applied to a large set of inputs produces errors for a large fraction of these inputs [198]. UAPs reveal systemic vulnerabilities in the target models and represent a significant risk, as they reduce the effort for the attacker to create adversarial examples. Successful UAPs have been generated for computer vision and object detection [43, 268, 89, 177], perceptual ad-blocking [291], and LiDAR-based object detection [48, 292]. However so far there has been little no exploration of problem-space UAP attacks against machine learning malware classifiers.

UAPs are perturbations that are universally effective across many inputs...

In malware development, we can envision a profit-motivated adversary such as a Malware-as-a-Service (MaaS) provider [146, 165, 306] with two objectives:

O1. They aim to *maximize* the amount of malware that can be made undetectable, increasing revenue.

O2. They aim to *minimize* the cost of making a single malware undetectable, reducing expenditure.

...particularly alluring to profit-motivated attackers as they can be generated once and reused multiple times.

From these objectives it is clear why UAPs are a natural choice of attack strategy: UAPs are *scalable*, amortizing the cost of generating a perturbation over the total number of evasive malware that it produces. Additionally, in the preceding sections we have proposed

and evaluated a scalable input-specific problem-space attack with properties aligned with these objectives. Here we see if the same methodology can be adapted to produce effective realizable UAPs.

4.6.1 *Feature-Space UAPs for Malware*

First we present a motivational experiment to demonstrate that malware classifiers are vulnerable to UAPs crafted in the feature space—that is, without considering the set of problem-space constraints which restrict how the attacker can mutate an input object.

Feature-space attacks may be unrealistic from a practical perspective, but their existence is a necessary condition for the existence of problem-space attacks (see Theorem 1).

Returning to the Android dataset described in Section 4.5.1, we create a random split with 60% of examples used for training and 40% for testing.

We again target the Drebin SVM [19] as well as a *Deep Neural Network (DNN)*.[3] The DNN has the architecture $5,000 \times 1,024 \times 512 \times 1$, and uses Leaky ReLU activation functions for the hidden layers (with negative slope equal to 0.1) and a sigmoid activation function for the output layer. We include Dropout at 0.1 to reduce overfitting and use the Adam optimizer [152] with learning rate equal to 10^{-3}.

We assess the robustness of the SVM and the DNN classifiers against input-specific and UAP attacks under perfect knowledge settings. Against the SVM we select the features with the most negative weights, *i.e.,* we select the top-L_0 features that are most indicative of goodware. For the UAP attack we take into account the contribution across all inputs.

We assess the robustness of SVM- and DNN-based malware detectors...

Against the DNN we apply the attack proposed by Grosse et al. [119] (derived from Papernot et al. [213]), which relies on the recursive computation of the Jacobian, searching at each step for the feature that maximizes the change in output in the desired direction (*i.e.,* towards evasion). For the UAP attack we propose a method where we select the most salient features computed by the Jacobian averaged over the malware examples in the test set. Given the binary feature space, we define the attacker's feature-space constraints in terms of the L_0-norm, i.e., the number of features that the attacker can modify, exploring values from $L_0 = 1$ to 20 and that the attacker can only *add* features, in order to preserve malicious semantics, *i.e.,* the attacker can only change features from 0 to 1 but not from 1 to 0. For the UAP attack, the effective change in the number of features that are set to 1 after the attack is at most L_0 for each input, *i.e.,* some of the features for these inputs may already be set to 1, and thus, the UAP does not change their value.

To quantify the success of the UAPs, we use the *Universal Evasion Rate* (UER) to measure the universality of each perturbation. UER is

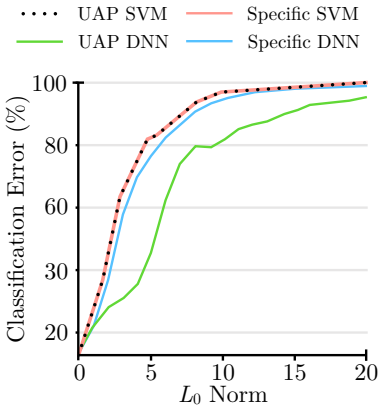


Figure 4.6: Input-specific vs. UAP perfect knowledge attacks against SVM and DNN in the feature space.

[3] We introduce a DNN baseline here as later we will try and to generate a more robust variant through adversarial training.

computed over a set of inputs $\mathcal{X}$ and defined as:

$$\text{UER} = \frac{|\{x \in \mathcal{X} : \arg\max g(x+\delta) \neq y \in \mathcal{Y}\}|}{|\mathcal{X}|}. \tag{4.15}$$

That is, UER denotes the fraction of inputs in $\mathcal{X}$ for which the classifier outputs an error when the UAP δ is applied.

...showing that both are weak against feature-space UAP attacks...

Figure 4.6 reports the classification error of the adversarial malware at different attack strengths (including when the malware is not manipulated, *i.e.*, $L_0 = 0$)—this is UER for the UAPs and simply the evasion rate for input-specific attacks. We observe that for $L_0 = 20$, the effectiveness of both UAP and input-specific attacks is above 95% in all cases, and in some cases achieves effectiveness close or equal to 100%.

Most importantly, we observe that the effectiveness of the UAP attacks is comparable to those of the input-specific attacks, especially for the SVM, where the results are almost identical.

For unprotected models, the extra capacity of the DNN compared to the linear classifier provides only marginal improvements in robustness that are not relevant from a practical perspective. Most importantly, our results show the importance of UAP attacks against malware classifiers: they achieve comparable effectiveness compared to their input-specific counterparts, but pose a significantly higher risk, as the same perturbation generalizes across many malware examples.

...justifying an exploration of whether problem-space UAP attacks are possible.

These results justify the attack methodology considered in the following sections, where we show it is also possible to generate very effective UAP attacks in the problem space, which pose a significant and real threat.

4.6.2 *Problem-Space UAPs for Malware*

Motivated by the results of feature-space UAP attacks in Section 4.6.1, here we show the feasibility of generating *problem-space* UAPs to realize real-world evasive malware.

Target Model. We target the Drebin SVM, using a random stratified split with 33% hold out for testing, partitioning the dataset into 101,596 and 50,041 examples for training and test, respectively. As before we avoid splitting the dataset temporally in order to evaluate the attacks in the *absence of concept drift*, to not overestimate the UAP success rate. We want to simulate a MaaS scenario in which an adversary is interested in *reusing* UAPs on future examples which they may not yet have access to. Therefore to ensure the UAPs do not overfit to the training set and will transfer to new malware examples, we set aside 10% (10,160) of the training set as the *exploration set* for the UAP search. As the final test set, we consider all true positive malware examples detected by the trained classifier (4,503 examples). On the non-adversarial (clean) test data the model achieves an AUC-ROC of 0.981 and 0.855 TPR at 1% FPR.

By adapting our previous attack to maximize UER...

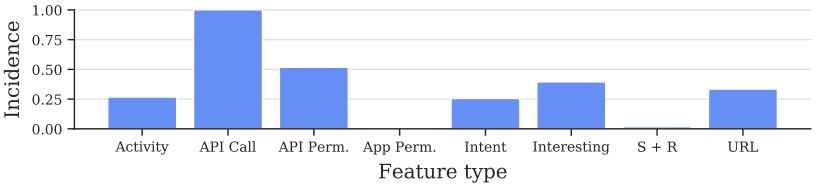


Figure 4.7: Relative incidence of feature perturbations, grouped by type, induced by the most effective individual transformations (UER $\geq$ 90%).

Available Transformations. We apply our attack as before, extracting 1,395 problem-space gadgets, based on features considered important with respect to benign examples in our exploration set, to obtain the final set of available transformations $\mathcal{T} = \{ t_0, \ldots, t_{1394} \}$, where t_i denotes the injection of gadget i into a given malware. Note that none of the transformations are capable of removal, only addition (*i.e.*, setting a feature value to 1).

UAP Search. As before, gadgets are selected greedily based on their total benign contribution (*i.e.*, considering side effects) and added until the decision score of the host malware is sufficiently benign. However, here we alter the search strategy to consider the UER across all true positive malware examples. We iteratively apply all possible transformations, at each step selecting the one maximizing UER across all true positive malware in the *exploration set*, until either the maximum length for the transformation sequence **T** is reached, 100% UER is reached, or no remaining transformations can increase UER. We consider a maximum sequence length of ten.

...we can discover strong UAP-based attacks in the problem-space.

Results. The attack is very successful, with the strongest UAP discovered using the exploration set producing 4,413 evasive variants on the test set after a single transformation (98% UER) and achieving 100% UER after two transformations.

Perturbation Analysis. We can now use the UAPs to better understand the weaknesses of the classifier by examining the nature of the feature-space perturbations induced by these strong transformations. Figure 4.7 shows the relative incidence of features, grouped by feature type, across the highly effective transformations (*i.e.*, with UER $\geq$ 90%). The most common feature types perturbed by the UAPs are related to API calls, with API calls perturbed by all transformations, API-related permissions perturbed by half, and a special category of "interesting" API calls being the third most common. However, the *individual* features which occur consistently across all of the top transformations are Activities, such as `activities::CloudAndWifiBaseActivity` (which is present in all but two of these transformations).

Although we reiterate that L_p norm constraints on the perturbation are not appropriate for problem-space attacks as the object can be modified arbitrarily so long as the problem-space constraints are not violated, it is still worth examining the size of the L_0 distortion induced by each transformation given how strong they appear to be individually.

Figure 4.8 shows the distribution of L_0 perturbation sizes, with a

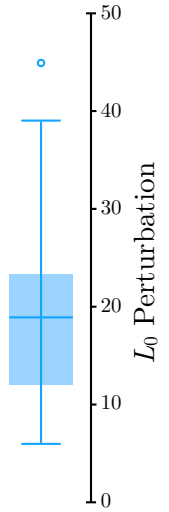


Figure 4.8: Distribution of L_0 norm perturbations (*i.e.*, number of binary features added) induced by the most effective individual transformations (UER $\geq$ 90%).

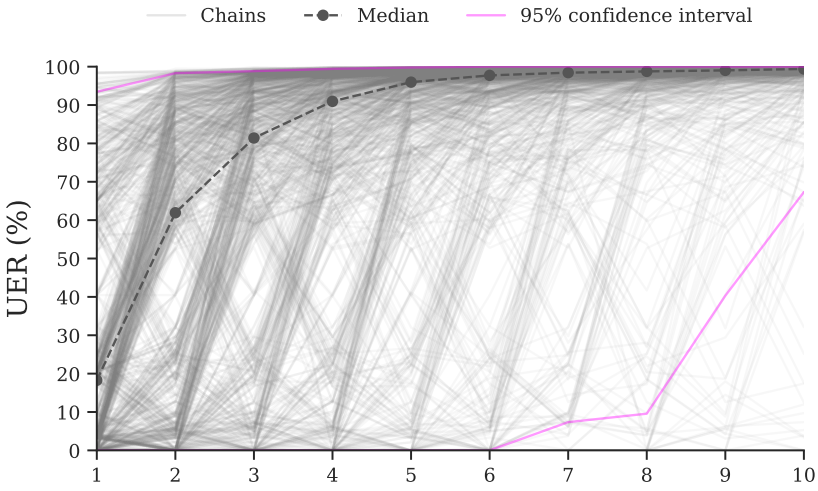


Figure 4.9: Limited knowledge attack against Drebin. UER for 1000 random transformation chains up to length 10. Relatively few transformations are required to achieve a high UER, highlighted by the median at each stage.

mean and median of 18.5 and 19, respectively. To provide perspective, the L_0 perturbation induced by the strongest transformation chain is 19; the mean and median L_0 norms of the overall dataset are 50 and 49, respectively.

We observe that the Android setting is particularly weak to strong problem-space attacks...

We observe that the Android domain appears naturally amenable to powerful attacks. The attacker is able to directly manipulate the bytecode, with established program analysis tools such as Soot [295] and FlowDroid [21] making specific alterations relatively accessible. Additionally, it is possible to choose gadgets from the ice-box such that the UER is monotonic with respect to gadget injection—where there is no risk of a transformation reducing the evasiveness of the transformation chain.

4.6.3 *Limited Knowledge Setting*

Having demonstrated the feasibility of problem-space UAPs in the perfect knowledge setting, here we look to see how effective UAPs might be in a setting where the attacker has only limited knowledge.

...and even limited knowledge attacks with random search can be successful.

Figure 4.9 shows the results from a naïve limited knowledge attack against the Android classifier, in which $\mathbf{T}$ is constructed by selecting gadgets at random. Each line depicts the UER produced by one of 1,000 transformation chains, tested at each stage of construction. The attack still appears to be extremely potent, with chains at length 5 achieving a median UER above 90%.

4.6.4 *Improving Robustness to UAPs*

A promising mitigation against adversarial examples is robust training [179, 58, 59], and in particular *adversarial training* [112, 181, 159]: the introduction of evasive examples into the training process to adjust the decision boundary to cover pockets of adversarial space close to legitimate examples. However, uniformly apply-

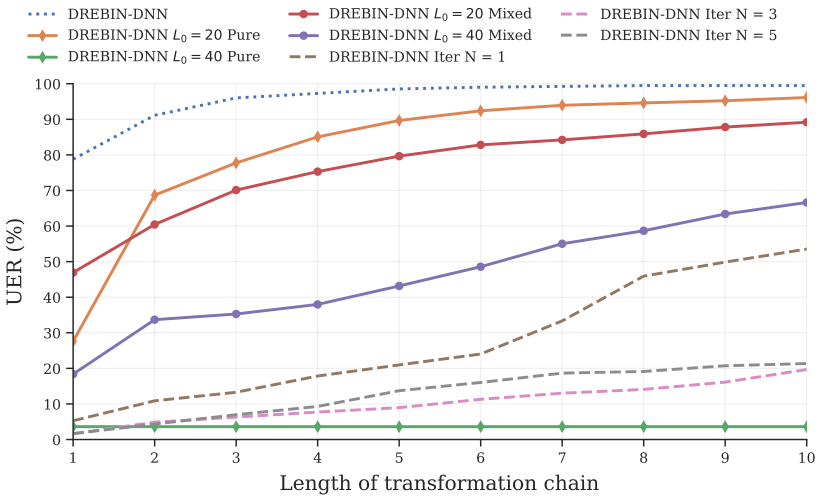


Figure 4.10: Adaptive attacks against hardened Drebin-DNN classifiers showing increasing UER at each stage of the transformation chain. Corresponding performance on clean data is shown in Table 4.2.

ing adversarial training to all regions close to the decision boundary can greatly alter the classifier, such that performance suffers on goodware or previously detected malware. Moreover, effort is wasted securing regions of the feature space which do not intersect with the feasible problem-space region of realizable attacks.

We hypothesize that an adversarial training variant based on problem-space UAPs...

Our UAPs show that the classifier has specific weaknesses against certain feature types (*e.g.,* API calls), so we posit that rather than applying adversarial training indiscriminately, we can instead use our UAPs to 'patch' the model against the specific toolkit of available transformations (here our ice-box). This would significantly raise the bar for attackers, forcing them to obtain a new set of transformations which may not even be possible.

Our process can be broadly defined as follows. *i) Generate problem-space UAPs* using a greedy search on the exploration set to calculate the strongest transformation chain, using the toolkit of available transformations, to quantify the model's initial robustness. *ii) Adversarially train the model*, by fine-tuning the model on newly generated UAPs as an additional step of the training process. *iii) Evaluate the robust models* considering an *adaptive attacker*, by performing a fresh search for UAP attacks. We focus on the effectiveness of the UAP attack in terms of UER, and the performance loss incurred on clean data in terms of AUC, and TPR at a fixed FPR of 1%.

...will be able to 'patch' the model against the attacker's set of available transformations.

Models. We hypothesize that the Drebin model will not be receptive to adversarial training, as the linear hyperplane will not be flexible enough to adapt to the adversarial inputs, *i.e.,* it will begin to 'forget' patterns of adversarial inputs seen earlier in the training process. To test this, we additionally apply our defense to the non-linear DNN originally described in Section 4.6.1 which we hereby refer to as Drebin-DNN.

During each of the the last N epochs of the training procedure, at the start of each minibatch, we apply our attack procedure to the partially trained model and search for the most effective UAP transformation sequence, *i.e.,* the UAP that maximizes UER across

	Model	AUC-ROC	TPR at 1% FPR	UER_1	UER_4	UER_{10}
Undefended	Drebin	0.981	0.855	98.7%	100%	100%
	Drebin-DNN	0.992	0.900	78.8%	97.3%	99.5%
Feature-space defenses	Drebin-DNN $L_0 = 20$ Pure	0.989	0.843	27.6%	85.1%	96.1%
	Drebin-DNN $L_0 = 40$ Pure	0.903	0.347	3.6%	3.6%	3.6%
	Drebin-DNN $L_0 = 20$ Mixed	0.990	0.872	46.9%	75.3%	89.2%
	Drebin-DNN $L_0 = 40$ Mixed	0.990	0.877	18.4%	38.0%	66.6%
Problem-space defenses	Drebin Iter $N = 1$	0.978	0.775	23.0%	70.4%	95.7%
	Drebin Iter $N = 3$	0.978	0.766	21.0%	47.0%	87.0%
	Drebin Iter $N = 5$	0.978	0.761	17.4%	35.1%	82.6%
	Drebin-DNN Iter $N = 1$	0.990	0.874	5.3%	17.9%	53.5%
	Drebin-DNN Iter $N = 3$	0.990	0.871	1.6%	7.7%	19.7%
	Drebin-DNN Iter $N = 5$	0.990	0.872	1.7%	9.3%	20.4%

Table 4.2: Comparison of problem-space vs. feature-space defenses, showing performance on clean examples (AUC-ROC, TPR) and robustness against an adaptive attacker (UER at $|\mathbf{T}|$ of 1, 4, and 10).

all true positive examples in the minibatch. For our problem-space defense, we fine-tune on the problem-space UAPs for the last $N = 1$, 3, and 5 epochs of training.

Baselines. As a baseline, we repeat the perfect knowledge problem-space attack against Drebin and Drebin-DNN (Table 4.2). For Drebin-DNN we also compare against a number of defenses obtained by generating adversarial examples in the feature space instead of the problem space. These defenses take two parameters: the L_0 constraint on the perturbation size and the percentage of adversarial examples to include during the adversarial training procedure. The second parameter is to prevent TPR from degrading from the model losing the ability to identify clean malware by being fine-tuned on adversarial examples alone. For the L_0 constraints we consider 20 and 40, and for the percentage of adversarial examples we consider a *pure* setting (100%) and a *mixed* setting where clean and adversarial malware examples are interleaved (50%).

The results confirm our defense is effective, producing a more focused adversarial training with a greater trade-off between robustness and clean-data accuracy.

Results. In Table 4.2 we report the UER of the adaptive white box attacks against each defense (also depicted in Figure 4.10). The close results for $N = 3$ and $N = 5$ appear to show an upper bound in the robustness gained, so it is likely that further epochs will result in diminishing returns. The results also confirm our hypothesis that the linear model is not as amenable to adversarial training as the nonlinear model. The linear Drebin model shows a larger performance loss on the clean examples compared to the other models (except for $L_0 = 40$ Pure), and while the robustness is improved for small sequences, UER for the sequences of length 10 is $> 80\%$.

Overall, the defense that provides the largest improvement in robustness is $L_0 = 40$ Pure, which is the most aggressive feature-space defense we consider, however it comes at a significant performance cost on non-adversarial examples. $L_0 = 40$ Mixed offers a better trade-off with a fairly large increase in robustness without the performance loss. The other feature-space defenses retain their performance on the non-adversarial examples, but do not show a significant gain in robustness.

However, our approach demonstrates an even more advanta-

geous trade-off than $L_0 = 40$ Mixed, with a similarly small performance loss on clean data, but far greater gains in robustness, reducing the maximum UER of length 10 chains from 99.5% to ~20%. This supports our hypothesis that integrating problem-space knowledge into the adversarial training process can result in a more focused and effective defense that specifically 'patches' vulnerabilities of interest.

4.7 Discussion on Attacks and Results

We provide some deeper discussion on the results of our novel problem-space attack.

We conclude that evasion of Android malware detectors is practically feasible...

Android Attack Effectiveness. We conclude that it is practically *feasible* to evade the Android malware classifier DREBIN [19] and its hardened variant, Sec-SVM [77], and that we are able to automatically generate realistic and inconspicuous evasive adversarial applications, often in less than 2 minutes. This shows for the first time that it is possible to create realistic adversarial applications at scale. Additionally the attack can be adapted to generate UAPs, scalable adversarial perturbations that transfer across many inputs, suitable for the MaaS threat model.

Obfuscation. It could be argued that traditional obfuscation methods can be used to simply hide malicious functionality. The novel problem-space attack in this work evaluates the feasibility of an "adversarial-malware as a service" scenario, where the use of mass obfuscation may raise the suspicions of the defender; for example, antivirus companies often classify samples as malicious simply because they utilize obfuscation or packing [4, 293]. Moreover, some other analysis methods combine static and dynamic analysis to prioritize evaluation of code areas that are likely obfuscated [*e.g.*, 168]. On the contrary, our transformations aim to be fully inconspicuous by adding only legitimate benign code and, to the best of our knowledge, we do not leave any relevant artifact in the process. While the effect on problem-space constraints may differ depending on the setting, attack methodologies such as ours and traditional obfuscation techniques naturally complement each other in aiding evasion and, in the program domain, code transplantation may be seen as a tool for developing new forms of inconspicuous obfuscation [92].

...and can be complemented by traditional obfuscation techniques.

Defense Directions Against Our Attacks. A recent promising direction by Incer et al. [133] studies the use of *monotonic classifiers*, where adding features can only increase the decision score (*i.e.*, an attacker cannot rely on adding more features to evade detection); however, such classifiers require non-negligible time towards manual feature selection (*i.e.*, on features that are harder for an attacker to change), and—at least in the context of Windows malware [133]—they suffer from high false positives and an average reduction in detection rate of 13%.

Moreover, we remark that we decide to add goodware gadgets to malware for practical reasons: the opposite transplantation would be immediate to do if a dataset with *annotated* malicious bytecode segments were available. As part of future work we aim to investigate whether it would still be possible to evade monotonic classifiers by adding only a minimal number of malicious slices to a benign application.

We hypothesize that monotonic classifiers may be promising to defend against addition-only attacks such as ours...

Defenses Against Problem-Space Attacks. Unlike settings where feature and problem space are closely related (*e.g.*, images and audio), limitations on feature-space L_p perturbations are often insufficient to determine the risk and feasibility of an attack in the real world. Our novel problem-space formalization (Section 4.3) paves the way to the study of *practical* defenses that can be effective in settings which lack an inverse feature mapping. Simulating and evaluating attacker capabilities in the problem space helps define realistic threat models with more constrained modifications in the feature space—which may lead to more robust classifier design. Our Android evasion attack (Section 4.4) demonstrates for the first time that it is *feasible* to evade feature-space defenses such as Sec-SVM in the problem-space—and to do so *en masse*.

...but emphasize that defenses must consider realistic problem-space constraints.

4.8 Related Work

Adversarial Machine Learning. Adversarial ML attacks have been studied for more than a decade [34]. These attacks aim to modify objects either at training time (*poisoning* [277]) or at test time (*evasion* [37]) to compromise the confidentiality, integrity, or availability of a machine learning model. Many formalizations have been proposed in the literature to describe feature-space attacks, either as optimization problems [37, 51] (see also Section 4.3.1 for details) or game theoretic frameworks [72].

Problem-Space Attacks. Recently, research on adversarial ML has moved towards domains in which the feature mapping is not invertible or not differentiable. Here, the adversary needs to modify the objects in the problem space (*i.e.*, input space) without knowing exactly how this will affect the feature space. This is known as the *inverse feature-mapping* problem [131, 37, 231]. Many works on problem-space attacks have been explored on different domains: text [169, 10], PDFs [185, 184, 164, 73, 318], Windows binaries [154, 232, 240], Android apps [77, 119, 321], NIDS [100, 16, 17, 65], ICS [331], and Javascript source code [231]. However, each of these studies has been conducted empirically and followed some inferred best practices: while they share many commonalities, it has been unclear how to compare them and what are the most relevant characteristics that should be taken into account while designing such attacks. Our formalization (Section 4.3) aims to close this gap, and we show how it can be used to describe representative feature-space and problem-space attacks from the

literature (Section 4.3.3).

Adversarial Android Malware. We also propose a novel adversarial problem-space attack in the Android domain (Section 4.4); our attack overcomes limitations of existing proposals, which are evidenced through our formalization. The most related approaches to our novel attack are on attribution [231], and on adversarial malware generation [321, 240, 119]. Quiring et al. [231] do *not* consider malware detection, but design a set of simple mutations to change the programming style of an application to match the style of a target developer (*e.g.,* replacing *for* loops with *while* loops). This strategy is effective for attribution, but is insufficient for malware detection as altering stylometric properties alone would not evade a malware classifier which captures program semantics. Moreover, it is not feasible to define a hardcoded set of transformations for all possible semantics—which may also leave artifacts in the mutated code. Conversely, our attack relies on automated software transplantation to ensure plausibility of the generated code and avoids hardcoded code mutation artifacts.

Grosse et al. [119] perform minimal modifications that preserve semantics, and only modify single lines of code in the Manifest; but these may be easily detected and removed due to unused permissions or undeclared classes. Moreover, they limit their perturbation to 20 features, whereas our problem-space constraints represent a more realistic threat model.

Yang et al. [321] propose a method for adversarial Android malware generation. Similarly to us, they rely on *automated software transplantation* [28] and evaluate their adversarial attack against the DREBIN classifier [19]. However, they do not formally define which semantics are preserved by their transformation, and their approach is extremely unstable, breaking the majority of apps they mutate (*e.g.,* they report failures after 10+ modifications on average—which means they would likely not be able to evade Sec-SVM [77] which on average requires modifications of 50+ features). Moreover, the code is unavailable, and the paper lacks details required for reevaluating the approach, including any clear descriptions of preprocessing robustness. Conversely, our attack is resilient to the insertion of a large number of features (Section 4.5), preserves dynamic app semantics through opaque predicates (Section 4.4.3), and is resilient against static program analysis (Section 4.4.4).

Rosenberg et al. [240] propose a black-box adversarial attack against Windows malware classifiers that relies on API sequence call analysis—an evasion strategy that is also applicable to similar Android classifiers. In addition to the limited focus on API-based sequence features, their problem-space transformation leaves two major artifacts which could be detected through program analysis: the addition of no-operation instructions (*no-ops*), and patching of the import address table (IAT). Firstly, the inserted API calls need to be executed at runtime and so contain individual no-ops hardcoded by the authors; intuitively, they could be detected and removed

by identifying the tricks used by attackers to perform no-op API calls (*e.g.,* reading 0 bytes), or by filtering the "dead" API calls (*i.e.,* which did not perform any real task) from the dynamic execution sequence before feeding it to the classifier. Secondly, to avoid requiring access to the source code, the new API calls are inserted and called using IAT patching. However, all of the new APIs must be included in a separate segment of the binary and, as IAT patching is a known malicious strategy used by malware authors [88], IAT calls to non-standard dynamic linkers or multiple jumps from the IAT to an internal segment of the binary would immediately be identified as suspicious. Conversely, our attack does not require hardcoding and by design is resilient against traditional non-ML program analysis techniques.

4.9 Summary

Since the seminal work that evidenced intriguing properties of neural networks [279], the community has become more widely aware of the brittleness of machine learning in adversarial settings [34].

To better understand real-world implications across different application domains, we propose a novel formalization of problem-space attacks as we know them today, that enables comparison between different proposals and lays the foundation for more principled designs in subsequent work. We uncover new relationships between feature space and problem space, and provide necessary and sufficient conditions for the existence of problem-space attacks. Our novel problem-space attack shows that automated generation of adversarial malware at scale is a realistic threat—taking on average less than 2 minutes to mutate a given malware example into a variant that can evade a hardened classifier. Additionally we have demonstrated the feasibility of using our attack to generate problem-space UAPs, scalable adversarial perturbations that transfer across many inputs, which can be used by Malware-as-a-Service providers to make many malware evasive at low cost.

Part III:

Detection in a Hostile Environment

> The function of wisdom is to
> discriminate between good and evil.
>
> Marcus Tullius Cicero

5 *Limiting Experimental Bias in ML for Security*

ADVERSARIAL INTERACTIONS and the hostile environment they produce, as discussed in Part II, can destabilize security detection systems. As a result, performing fair and informative experiments in such a dynamic and volatile environment is very challenging.

For example, as attackers drive the malicious class to evolve over time, bias can be introduced if knowledge about that evolution is not correctly handled during training and test phases. Similarly, not accounting for the typically low prevalence of the malicious class in the general population can result in inflated performance results.

To begin, this chapter examines how to design and evaluate machine learning-based security detectors while accounting for the specific nature of the hostile environment, using malware detectors as a case study. Once a stable and consistent evaluation is possible, we can compare potential mitigations against concept drift and in Chapter 6 we will dive deeper into a classification-with-rejection approach for detecting and quarantining drifting examples.

5.1 KEY INSIGHTS

For reference, this chapter provides the following contributions:

- We identify *temporal* bias associated with incorrect train-test splits (Section 5.4.2) and *spatial* bias related to unrealistic assumptions in dataset distribution (Section 5.4.3).
- We experimentally verify on a dataset of 129K apps (with 10% malware) that, once sources of bias are removed, performance can decrease up to 50% in practice (Section 5.4.1) for two well-known Android malware classifiers, DREBIN [19] (ALG1) and MAMADROID [187] (ALG2).
- We propose novel building blocks for the more robust evaluation of malware classifiers: a set of spatio-temporal constraints to be enforced in experimental settings (Section 5.5.1); a new metric, AUT, that captures a classifier's robustness to time decay in a single number and allows for the fair comparison of different

algorithms (Section 5.5.2); and a novel tuning algorithm that empirically optimizes the classification performance, when malware represents the minority class (Section 5.5.3).

- We compare the performance of ALG1 [19], ALG2 [187] and DL [118] (a deep learning-based approach), and show how removing sources of bias can lead to counterintuitive performance results (Section 5.6).
- We implement our methodology as an open-source Python library, TESSERACT, to aid in the design and execution of fair evaluations, and further demonstrate how our findings can be used to evaluate performance-cost trade-offs of solutions to mitigate time decay such as active learning (Section 5.7).
- To more broadly inspect the impact of spatio-temporal bias, we assess their pervasiveness in recent literature beyond the Android malware domain (Section 5.8). For completeness, we also describe a general set of *pitfalls* afflicting machine learning-based security evaluations, their prevalence in recent literature, and recommendations for avoiding them (Section 5.10).

The content of this chapter has been previously presented in the following publications:

- Pendlebury F.*, Pierazzi F.*, Jordaney R., Kinder J., Cavallaro L. Enabling Fair ML Evaluations for Security. In *Proc. of the ACM Conference on Computer and Communications Security (CCS) (poster)*. 2018.
- Pendlebury F.*, Pierazzi F.*, Jordaney R., Kinder J., Cavallaro L. TESSERACT: Eliminating Experimental Bias in Malware Classification Across Space and Time. In *Proc. of the USENIX Security Symposium*. 2019.
- Arp, D., Quiring E., Pendlebury F., Warnecke A., Pierazzi F., Wressnegger C., Cavallaro L., Rieck K. Dos and Don't of Machine Learning in Computer Security. To appear in *Proc. of the USENIX Security Symposium*. 2022.

5.2 OVERVIEW

Machine learning-based malware detection approaches have reported tantalizingly high performance figures across a wide range of domains including Windows malware [71, 281, 188], PDF malware [271, 184], malicious URLs [274, 167], malicious JavaScript [236, 70], and Android malware [19, 187, 118]. With results approaching 100% accuracy, it seems malware should be a problem of the past.

Prior work on ML-based malware detection has reported almost perfect results...

However, security settings have specific properties which need to be accounted for. For example, as we have shown, malware classifiers operate in a hostile, dynamic environment. As malware

evolves and new variants and families appear over time, prediction quality decays [139], therefore, temporal consistency matters for evaluating the effectiveness of a classifier. Some experimental setups may erroneously allow a classifier to train on what is effectively future knowledge, inflating the reported results [6, 194].

It turns out that sources of bias have affected evaluations throughout the security community, affecting multiple security domains. In this chapter we focus on the Android malware domain as a case study as there exists an endemic issue where Android malware classifiers [*e.g.*, 19, 276, 187, 109, 324, 74, 323, 118] are not evaluated in settings representative of real-world deployments. Our reasons for focusing on Android are due to the availability of (*a*) a public, large-scale, and timestamped dataset (AndroZoo [8]) and (*b*) algorithms that are feasible to reproduce (where all [187] or part [19] of the code has been released).

...particularly for Android malware detection...

We identify experimental bias in two dimensions, *space* and *time*. *Spatial bias* refers to unrealistic assumptions about the ratio of goodware to malware in the data. The ratio of goodware to malware is domain-specific, but it must be enforced consistently during the test phase to mimic a realistic scenario. For example, measurement studies on Android suggest that most apps in the wild are goodware [173, 114], whereas for (desktop) software download events most URLs are malicious [183, 234].

Temporal bias refers to temporally inconsistent evaluations which integrate future knowledge about the test objects into the training phase [6, 194] or create unrealistic settings. This problem is exacerbated by families of closely related malware, where including even one variant in the training set may allow the algorithm to identify many variants in the test set.

...but these results are largely inflated by experimental bias.

We believe that the pervasiveness of these issues is due to two main reasons: first, possible sources of evaluation bias are not common knowledge; second, accounting for time complicates the evaluation and does not allow a comparison to other approaches using headline evaluation metrics such as the F_1-Score or AUC. Here we address these issues by systematizing evaluation bias for Android malware classification and providing new constraints for sound experiment design along with new metrics and tool support.

This study continues a line of investigation started by prior work on challenges and experimental bias in security evaluations [241, 267, 296, 6, 194, 23]. The *base-rate fallacy* [23] describes how evaluation metrics such as TPR and FPR are misleading in intrusion detection, due to significant class imbalance (*i.e.*, most traffic is benign); in contrast, we identify and address experimental settings that give misleading results *regardless* of the adopted metrics—even when correct metrics are reported. Sommer and Paxson [267], Rossow et al. [241], and van der Kouwe et al. [296] discuss possible guidelines for sound security evaluations; but none of these works identify temporal and spatial bias, nor do they *quantify* the impact of errors on classifier performance. Allix et al. [6] and

We identify sources and effects of spatio-temporal bias...

Miller et al. [194] identify an initial temporal constraint in Android malware classification, but we show that even the results of recent work that followed their guidelines (*e.g.*, Mariconti et al. [187]) suffer from other forms of temporal and spatial bias (Section 5.6). To the best of our knowledge, this study is the first to identify and address these sources of bias with novel, actionable constraints, metrics, and tool support (Section 5.5).

We introduce a framework for fair evaluations, TESSERACT, that can assist the research community in producing comparable results, revealing counterintuitive performance, and assessing a classifier's prediction qualities in an industrial deployment (Section 5.9). TESSERACT also creates an opportunity to evaluate the extent to which spatio-temporal experimental bias affects security domains other than Android malware, and we encourage the security community to embrace its underpinning philosophy.

...and propose constraints, metrics, and tooling to support fair, realistic evaluations.

5.3 Initial Experimental Setup

This section describes the initial experimental setup with which we will assess experimental bias in the reference algorithms. As a case study, we focus on Android malware classification. In Section 5.3.1 we introduce the reference approaches evaluated, in Section 5.3.2 we discuss the domain-specific prevalence of malware, and in Section 5.3.3 we introduce the dataset used throughout the chapter.

> **Use of the term "bias":** We use *(experimental) bias* to refer to the details of an experimental setting that depart from the conditions in a real-world deployment and can have a misleading impact (*bias*) on evaluations. It should not be conflated with the classifier bias/variance trade-off [40] from traditional machine learning terminology.

5.3.1 *Reference Algorithms*

To assess experimental bias (Section 5.4), we consider two high-profile machine learning-driven techniques for Android malware classification, both published in top-tier security conferences. The first approach is Drebin (**Alg1**) [19], a linear support vector machine (SVM) on high-dimensional binary feature vectors engineered with a lightweight static analysis. The second approach is MaMaDroid (**Alg2**) [187], a Random Forest (RF) applied to features engineered by modeling caller-callee relationships over Android API methods as Markov chains.

We assess bias in the evaluations of Drebin [19] and MaMaDroid [187]...

We choose ALG1 and ALG2 as they build on different types of static analysis to generate feature spaces capturing Android application characteristics at different levels of abstraction; furthermore,

they use different machine learning algorithms to learn decision boundaries between benign and malicious Android apps in the given feature space. Thus, they represent a broad design space and support the generality of our methodology for characterizing experimental bias.

For a sound experimental baseline, we replicate the settings and experiments of ALG1 [19] (linear SVM with C=1) and ALG2 [187] (package mode and RF with 101 trees and max depth of 64) as described in the respective papers [19, 187], successfully reproducing the published results. Since on our dataset (described in Section 5.3.3) the ALG1 performance is slightly lower (~0.91 10-fold F_1-Score), we also reproduce the experiment on their original dataset [19], achieving their original performance of ~0.94 10-fold F_1-Score. We use SCIKIT-LEARN [221], with *sklearn.svm.LinearSVC* for ALG1 and *sklearn.ensemble.RandomForestClassifier* for ALG2. Since ALG1 and ALG2 adopt traditional ML algorithms, in Section 5.5 we also consider **DL** [118], a deep learning-based approach that takes as input the same features as ALG1 [19]. We include DL because the latent feature space of deep learning approaches can capture different representations of the input data [118], which may affect their robustness to time decay. We follow the guidelines from Grosse et al. [118] to reimplement DL with KERAS. The features given as initial input to the neural network are the same as ALG1. We replicate the best-performing neural network architecture of Grosse et al. [118], by training with 10 epochs and batch size equal to 1,000. To perform the training optimization, we use the stochastic gradient descent class *keras.optimizers.SGD* with the following parameters: *lr*=0.1, *momentum*=0.0, *decay*=0.0, *nesterov*=False. Some low-level details of the hyperparameter optimization were missing from the original paper [118]; we managed to obtain slightly higher F_1 performance in 10-fold setting likely because they have performed hyperparameter optimization on the Accuracy metric [40]—which is misleading in imbalanced datasets [23] where one class is more prevalent (*e.g.,* goodware, in Android).

...as well as a deep learning-based Android malware detector [118].

It speaks to the scientific standards of these papers that we were able to replicate the experiments; indeed, we would like to emphasize that we do not criticize them specifically. We use these approaches for our evaluation because they are available and offer stable baselines.

5.3.2 *Estimating in-the-wild Malware Ratio*

The proportion of malware in the dataset can greatly affect the performance of the classifier (Section 5.4), therefore, unbiased experiments require a dataset with a realistic ratio of malware to goodware; on an already existing dataset, this ratio may be enforced by, for instance, downsampling the majority class (Section 5.4.3).

Before constructing a dataset, a realistic prevalence for the malicious class must be estimated...

Each malware domain has its own, often unique, ratio of mal-

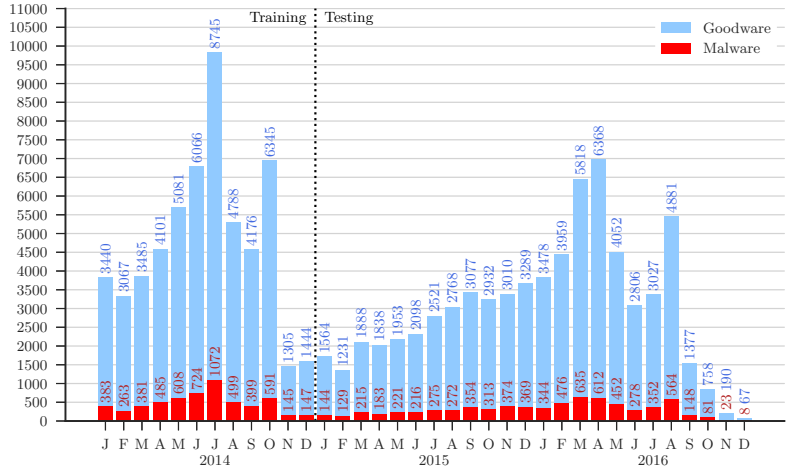


Figure 5.1: A stack histogram showing the monthly distribution of apps in our dataset of 129,728 Android applications (with average 10% malware) from 2014–2016.

ware to goodware typically encountered in the wild. First, it is important to know if the ratio is balanced, or if malware is the minority or majority class. For example, malware is the minority class in network traffic [23] and Android [173], but it is the majority class in binary download events [234].

On the one hand, the estimation of the percentage of malware in the wild for a given domain is a non-trivial task. On the other hand, measurement papers, industry telemetry, and publicly-available reports may all be leveraged to obtain realistic estimates.

In the Android landscape, malware represents 6%–18.8% of all apps, according to different sources: a key industrial player[1] reported the ratio as approximately 6%, whereas the AndRadar measurement study [173] reports around 8% of Android malware in the wild. Google's 2017 Android security report [114] suggests 6–10% malware, whereas an analysis of the metadata of the AndroZoo dataset [8] totaling almost 8M Android apps updated regularly, reveals an incidence rate of 18.8%.

[1] Information obtained through confidential emails with the authors.

The data suggests that, in the Android domain, malware is the minority class. For our experiments, we stabilize its percentage at 10% (a de-facto average across the various estimates), with per-month values between 8% and 12%. Settling on an average overall ratio of 10% Android malware also allows us to collect a dataset with a statistically sound number of per-month malware. An aggressive undersampling would have decreased the statistical significance of the dataset, whereas oversampling goodware would have been too resource intensive (Section 5.3.3).

...which for Android malware we determine to be ~10%.

5.3.3 *Dataset*

We consider samples from the AndroZoo [8] dataset, consisting of more than 8.5 million Android apps between 2010 and 2019: each is associated with a timestamp, and most apps include VirusTotal

metadata results. The dataset is kept updated by crawling from different markets (*e.g.,* more than 4 million apps from Google Play Store, and the remaining from markets such as Anzhi and AppChina). We use this dataset due to its size and timespan, allowing us to perform realistic space- and time-aware experiments.

We construct a dataset with apps from the AndroZoo [8] repository...

Goodware and malware. AndroZoo's metadata reports the number p of positive anti-virus reports on VirusTotal [115] for applications in the AndroZoo dataset. We chose $p = 0$ for goodware and $p \geq 4$ for malware, following Miller et al.'s [194] advice for a reliable ground-truth. About 13% of AndroZoo apps can be called grayware as they have $0 < p < 4$. We exclude grayware from the sampling as including it as either goodware or malware could disadvantage classifiers whose features were designed with a different labeling threshold, however we note that for the evaluation of new detection systems this can itself introduce a bias (Section 5.10).

Choosing apps. The number of apps we consider is affected by the feature extraction cost, and partly by storage space requirements (as the full AndroZoo dataset at the time of writing is more than 50TB). Extracting features for the whole AndroZoo dataset may take up to three years on our research infrastructure (three Dell PowerEdge R730 nodes, each with 2 x 14 cores in hyperthreading—in total, 168 vCPU threads, 1.2TB of RAM, and a 100TB NAS), so we decide to extract features from 129K apps (Section 5.3.2). We believe this represents a large dataset with enough statistical significance. To evaluate time decay, we choose a granularity of one month, and uniformly sample 129K AndroZoo apps in the period from Jan 2014 to Dec 2016 while enforcing an overall average of 10% malware (see Section 5.3.2)—with an allowed percentage of malware per month between 8% and 12%, to ensure some variability. Spanning over three years ensures 1,000+ apps per month (except for the last three months, where AndroZoo had crawled less applications). We consider apps up to Dec 2016 as VirusTotal results for 2017 and 2018 apps were mostly unavailable from AndroZoo at the time of the study; moreover, Miller et al. [194] empirically evaluated that antivirus detections stabilize after approximately one year so ending at Dec 2016 increases confidence in malware ground-truth labels.

...with apps from 2014–2016 and partitioned by month...

Dataset summary. The final dataset consists of 129,728 Android applications (116,993 goodware and 12,735 malware). Figure 5.1 reports a stack histogram showing the per-month distribution of goodware/malware in the dataset. For the sake of clarity, the figure also reports the number of malware and goodware in each bin. The training and test splits used in Section 5.4 are reported in Table 5.2; all the time-aware experiments in the remainder of this chapter are performed by training on 2014 and testing on 2015 and 2016 (see the vertical dotted line in Figure 5.1).

...totaling 116,993 goodware and 12,735 malware overall.

Work	Apps	Date Range	# Objects	Total	Violations
TESSERACT (this work)	Benign	Jan 2014 - Dec 2016	116,993	116,993	-
	Malicious		12,735	12,735	
ALG1 [19] & DL [118]	Benign	Aug 2010 - Oct 2012	123,453	123,453	C1
	Malicious		5,560	5,560	
ALG2 [187]	Benign	Apr 2013 - Nov 2013	5,879	8,447	(C1) C2 C3
		Mar 2016	2,568		
	Malicious	Oct 2010 - Aug 2012	5,560	35,493	
		Jan 2013 - Jun 2013	6,228		
		Jun 2013 - Mar 2014	15,417		
		Jan 2015 - Jun 2015	5,314		
		Jan 2016 - May 2016	2,974		

Table 5.1: Summary of dataset composition in this study (1st row) compared to prior work [19, 187, 118] (2nd and 3rd row).

For reference, Table 5.1 compares the composition of the dataset used in our study (1st row) to the datasets used in the biased evaluations for ALG1 [19], ALG2 [187], and DL [118] (2nd and 3rd row). Unless otherwise stated, we always evaluate ALG1, ALG2, and DL with the dataset in the first row, because we have built it to allow experiments without spatio-temporal bias.

5.4 Sources of Experimental Bias

In this section, we motivate our discussion of bias through an empirical assessment of ALG1 [19] and ALG2 [187] (Section 5.4.1). We then detail the sources of temporal (Section 5.4.2) and spatial bias (Section 5.4.3) that affect ML-based Android malware classification.

5.4.1 *Motivational Example*

We consider a motivational example in which we vary the sources of experimental bias to illustrate the problem. Table 5.2 reports the F_1-Score for ALG1 and ALG2 under various experimental configurations; rows correspond to different sources of temporal bias, and columns correspond to different sources of spatial bias. On the left-part of Table 5.2, we use squares (■/■) to show from which time frame training and test objects are taken; each represents six months (in the window from Jan 2014 to Dec 2016). Black squares (■) denote that samples are taken from that six-month time frame, whereas periods with gray squares (■) are not used. The columns on the right part of the table correspond to different percentages of malware in the training set *Tr* and the test set *Ts*.

We compare the original experimental settings [19, 187] to more realistic settings...

Table 5.2 shows that both ALG1 and ALG2 perform far worse in realistic settings (bold with blue background in the last row, for columns corresponding to 10% malware in the test set) than in settings similar to those presented in the original proposals [19, 187] (bold with red background) due to inadvertent experimental bias.

...showing that results are inflated due to experimental bias.

			% mw in test set Ts							
			10% (*realistic*)				90% (*unrealistic*)			
			% mw in training set Tr				% mw in training set Tr			
	Sample dates		10%	90%	10%	90%	10%	90%	10%	90%
Experimental setting	Training	Test	ALG1 [19]		ALG2 [187]		ALG1 [19]		ALG2 [187]	
10-fold CV	gw: ■■■■■■ mw: ■■■■■■	gw: ■■■■■■ mw: ■■■■■■	**0.91**	0.56	0.83	0.32	0.94	0.98	0.85	**0.97**
Temporally inconsistent	gw: □□□□■■ mw: □□□□■■	gw: ■■■■□□ mw: ■■■■□□	0.76	0.42	0.49	0.21	0.86	0.93	0.54	**0.95**
Temporally inconsistent gw/mw windows	gw: □□□□■□ mw: ■■□□□□	gw: □□□□□■ mw: □□■■□□	0.77	0.70	0.65	0.56	0.79	0.94	0.65	**0.93**
Temporally consistent (*realistic*)	gw: ■■□□□□ mw: ■■□□□□	gw: □□■■■■ mw: □□■■■■	**0.58**	**0.45**	**0.32**	**0.30**	0.62	0.94	0.33	0.96

Table 5.2: F_1-Scores showing impact of spatial (columns) and temporal (rows) bias. Values in red are results of (unrealistic) settings similar to the original papers [19, 187]; values in blue (last row) are results in realistic settings. Squares are six month data windows: black if included (■), gray if not (□).

Note: We clarify which similar settings of the original proposals [19, 187] we refer to in the cells with red background in Table 5.2. The paper of ALG2 [187] reports in the abstract performance "up to 99% F_1", which (out of the many settings they evaluate) corresponds to a scenario with 86% malware in both training and test, evaluated with 10-fold CV; here we round to 90% malware for a cleaner presentation (we have experimentally verified that results with 86% and 90% malware-to-benign class ratio are similar). ALG1's original paper [19] relies on hold-out by performing 10 random splits (66% training and 33% test). Since hold-out is almost equivalent to k-fold CV and suffers from the same spatio-temporal biases, for the sake of simplicity in this section we refer to a k-fold CV setting for both ALG1 and ALG2.

5.4.2 *Temporal Experimental Bias*

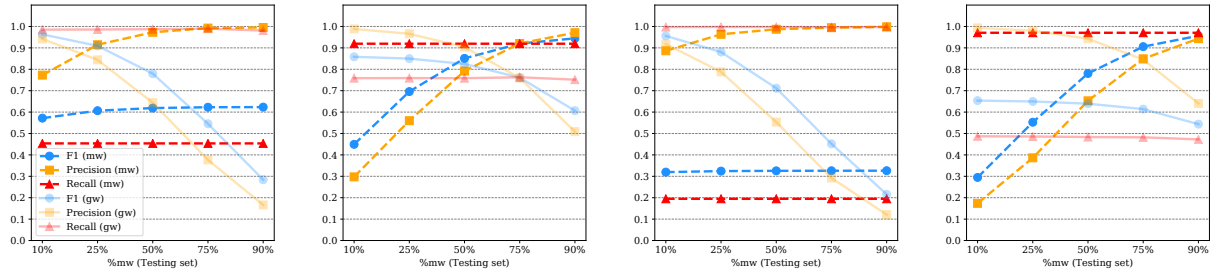
As discussed in previous chapters, *concept drift*, also known as *dataset shift* [199], is a problem that occurs in machine learning when a model becomes obsolete as the distribution of incoming data at test-time differs from that of training data, *i.e.*, when the assumption does not hold that data is independent and identically distributed (i.i.d.) [139]. *Time decay* is the decrease in model performance over time caused by concept drift.

Concept drift in malware, combined with similarities among malware within the same family, causes *k-fold cross validation* (CV) to be *positively biased*, artificially inflating the performance of malware classifiers [6, 194, 195].

Not accounting for drift in the data distribution...

K-fold CV is likely to include in the training set at least one sample of each malware family in the dataset, whereas new families will be unknown at training time in a real-world deployment. The all-black squares in Table 5.2 for 10-fold CV refer to each training/test fold of the 10 iterations containing at least one sample from each time frame.

The use of k-fold CV is widespread in malware classification re-



(a) ALG1: Train with 10% mw (b) ALG1: Train with 90% mw (c) ALG2: Train with 10% mw (d) ALG2: Train with 90% mw

Figure 5.2: *Spatial bias in testing*. For increasing % of test set malware, Precision increases and Recall remains the same; causing F_1-Score to rise.

search [195, 234, 276, 183, 71, 281, 188, 271, 324, 70]; while a useful mechanism to prevent overfitting [40] or estimate the performance of a classifier in the *absence* of concept drift when the i.i.d. assumption holds (see considerations in Section 5.6), it has been unclear how it affects the real-world performance of machine learning techniques with non-stationary data that are affected by time decay. Here, in the first row of Table 5.2, we quantify the performance impact in the Android malware domain.

The second row of Table 5.2 reports an experiment in which a classifier's ability to detect past objects is evaluated [6, 187]. Although this characteristic is important, high performance should be expected from a classifier in such a scenario: if the classifier contains at least one variant of a past malware family, it will likely identify similar variants. We thus believe that experiments on the performance achieved on the detection of past malware can be misleading; the community should focus on building malware classifiers that are robust against time decay.

...leads to temporal bias where models are anachronistically trained on data from the future...

In the third row, we identify a novel temporal bias that occurs when goodware and malware correspond to different time periods, often due to having originated from different data sources [*e.g.*, 187]. The black and gray squares in Table 5.2 show that, although malware test objects are posterior to malware training objects, the goodware/malware time windows do not overlap; in this case, the classifier may learn to distinguish applications from different time periods, rather than goodware from malware—again leading to artificially high performance. For instance, spurious features such as new API methods may be able to strongly distinguish objects simply because malicious applications predate that API.

...or learn to distinguish between new and old apps rather than malicious and benign behavior.

The last row of Table 5.2 shows that the realistic setting, where training is temporally precedent to testing, causes the worst classifier performance in the majority of cases. We present decay plots and a more detailed discussion in Section 5.5.

5.4.3 *Spatial Experimental Bias*

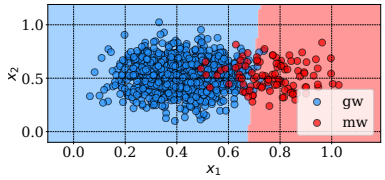
We identify two main types of spatial experimental bias based on assumptions on percentages of malware in test and training sets. All experiments in this section assume temporal consistency.

The model is trained on 2014 and tested on 2015 and 2016 (last row of Table 5.2) to allow the analysis of spatial bias without the interference of temporal bias.

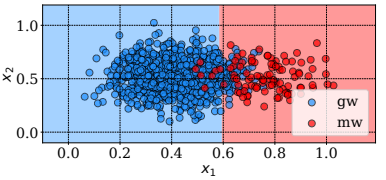
Spatial experimental bias in testing. The percentage of malware in the test distribution needs to be estimated (Section 5.3.2) and *cannot* be changed, if one wants results to be representative of in-the-wild deployment of the malware classifier. To understand why this leads to biased results, we artificially vary the test distribution. Figure 5.2 reports the performance (F_1-Score, Precision, Recall) when increasing the percentage of malware during testing on the x-axis. We change the percentage of malware in the test set by randomly downsampling goodware, so that the number of malware remains fixed throughout the experiments. For completeness, we report the two training settings from Table 5.2 with 10% and 90% malware, respectively.

Testing with unrealistic base rates will produce inflated results...

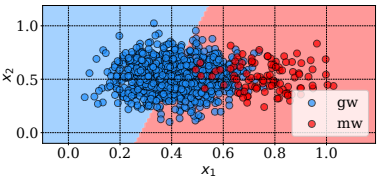
Let us first focus on the malware performance (dashed lines). All plots in Figure 5.2 exhibit constant Recall, and increasing Precision for increasing percentage of malware in the test set. Precision for the malware (mw) class—the positive class—is defined as $P_{mw} = {}^{TP}/_{(TP+FP)}$ and Recall as $R_{mw} = {}^{TP}/_{(TP+FN)}$. In this scenario, we can observe that TPs (*i.e.*, malware objects correctly classified as malware) and FNs (*i.e.*, malware objects incorrectly classified as goodware) do not change, because the number of malware does not increase; hence, Recall remains stable. The increase in number of FPs (*i.e.*, goodware objects misclassified as malware) decreases as we reduce the number of goodware in the dataset; hence, Precision improves. Since the F_1-Score is the harmonic mean of Precision and Recall, it goes up with Precision. We also observe that, inversely, the Precision for the goodware (gw) class—the negative class—$P_{gw} = {}^{TN}/_{(TN+FN)}$ decreases (see yellow solid lines in Figure 5.2), because we are reducing the TNs while the FNs do not change. This example shows how considering an unrealistic test distribution with more malware than goodware in this context (Section 5.3.2) positively inflates Precision and hence the F_1-Score of the malware classifier.



(a) Training 10% mw; Testing 10% mw



(b) Training 50% mw; Testing 10% mw

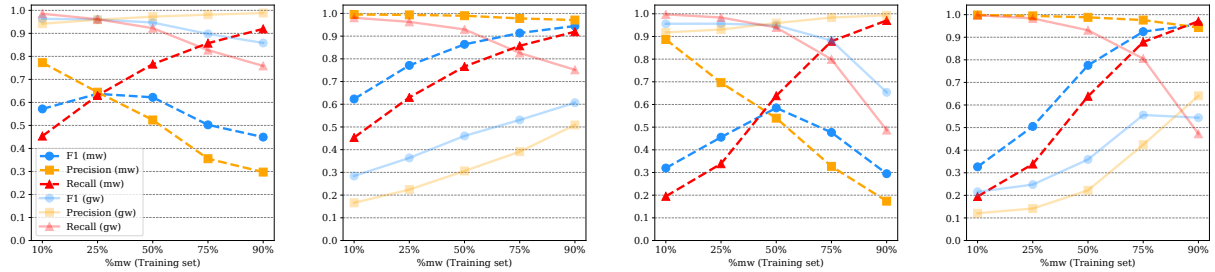


(c) Training 90% mw; Testing 10% mw

Figure 5.3: Example of training bias with Linear-SVM and two features. Test set is fixed at 10% malware but as rate in the training set increases, the decision boundary moves towards goodware, improving Recall but decreasing Precision.

Spatial experimental bias in training. To understand the impact of altering malware-to-goodware ratios in training, we now consider a motivating example with a linear SVM in a 2D feature space, with features x_1 and x_2. Figure 5.3 reports three scenarios, all with the same 10% malware in the test set, but with 10%, 50%, and 90% malware in training.

We can observe that with an increasing percentage of malware in training, the hyperplane moves towards goodware. More formally, it improves Recall of malware while reducing its Precision. The opposite is true for goodware. To minimize the overall error rate $Err = {}^{(FP+FN)}/_{(TP+TN+FP+FN)}$ (*i.e.*, maximize Accuracy), one should train the dataset with the same distribution that is expected



(a) ALG1: Test with 10% mw (b) ALG1: Test with 90% mw (c) ALG2: Test with 10% mw (d) ALG2: Test with 90% mw

Figure 5.4: *Spatial bias in training.* For increasing % of malware in the training, Precision decreases and Recall increases.

...but altering the class balance of the training set can be a legitimate way to tune the classifier.

in the test set. However, in this scenario one may have more interest in finding objects of the minority class (*e.g.*, "more malware") by improving Recall subject to a constraint on maximum FPR.

Figure 5.4 shows the performance for ALG1 and ALG2, for increasing percentages of malware in training on the X-axis; just for completeness (since one cannot artificially change the test distribution to achieve realistic evaluations), we report results both for 10% malware in testing and for 90% malware in testing, but we remark that in the Android setting we have estimated 10% malware in the wild (Section 5.3.2). These plots confirm the trend in our motivating example (Figure 5.3), that is, R_{mw} increases but P_{mw} decreases. For the plots with 10% malware in testing, we observe there is a point in which F_1-Score$_{mw}$ is at a maximum while the error for the goodware class is within 5%.

In Section 5.5.3, we propose a novel algorithm to improve the performance of the malware class according to the objective of the user (high Precision, Recall or F_1-Score), subject to a maximum tolerated error. Moreover, in Section 5.5 we introduce constraints and metrics to guarantee bias-free evaluations, while revealing counterintuitive results.

Remark on downsampling We choose to downsample goodware (gw) to achieve up to 90% of malware (mw) for testing because of the computational and storage resources required to achieve such a ratio by oversampling. This does not alter the conclusions of our analysis. Let us assume a scenario in which we keep the same number of goodware, and increase the percentage of mw in the dataset by oversampling mw. The precision ($P_{mw} = {}^{TP}/_{(TP+FP)}$) would increase because TPs would increase for any mw detection, and FPs would not change—because the number of gw remains the same; if training (resp. test) observations are sampled from a distribution similar to the mw in the original dataset (*e.g.*, new training mw is from 2014 and new test mw comes from 2015 and 2016), then Recall ($R_{mw} = {}^{TP}/_{(TP+FN)}$) would be stable—it would have the same proportions of TPs and FNs because the classifier will have a similar predictive capability for finding mw. Hence, if the number of mw in the dataset increases, the F_1-Score would increase as well, because Precision increases while Recall remains stable.

5.5 Space-Time Aware Evaluation

We now formalize how to perform an evaluation of an Android malware classifier free from spatio-temporal bias. We define a novel set of constraints that must be followed for realistic evaluations (Section 5.5.1); we introduce a novel time-aware metric, AUT, that captures in one number the impact of time decay on a classifier (Section 5.5.2); we propose a novel tuning algorithm that empirically optimizes a classifier's performance, subject to a maximum tolerated error (Section 5.5.3); finally, we introduce Tesseract and provide counterintuitive results through unbiased evaluations (Section 5.6). For reference, we report in the front matter of this thesis a table with all major symbols used in the remainder of this chapter.

5.5.1 *Evaluation Constraints*

Let us consider D as a labeled dataset with two classes: malware (positive class) and goodware (negative class). Let us define $s_i \in D$ as an *object* (*e.g.*, Android app) with timestamp $time(s_i)$. To evaluate the classifier, the dataset D must be split into a training dataset Tr with a time window of size W, and a test dataset Ts with a time window of size S. Here, we consider $S > W$ in order to estimate long-term performance and robustness to decay of the classifier. A user may consider different time splits depending on their objectives, provided each split has a significant number of samples. We emphasize that, although we have the labels of objects in $Ts \subseteq D$, all the evaluations and tuning algorithms *must* assume that labels y_i of objects $s_i \in Ts$ are unknown.

We formalize three rules for avoiding spatio-temporal bias:...

To evaluate performance over time, the test set Ts must be split into time-slots of size Δ. For example, for a test set time window of size $S = 2$ years, we may have $\Delta = 1$ month. This parameter is chosen by the user, but it is important that the chosen granularity allows for a statistically significant number of objects in each test window $[t_i, t_i + \Delta)$.

We now formalize three constraints that must be enforced when dividing D into Tr and Ts for a realistic setting that avoids spatio-temporal experimental bias (Section 5.4). While C1 was proposed in past work [6, 194], we are the first to propose C2 and C3—which we show to be fundamental in Section 5.6.

C1) Temporal training consistency. All the objects in the training must be *strictly* temporally precedent to the testing ones:

...training data should temporally precede test data...

$$time(s_i) < time(s_j), \forall s_i \in Tr, \forall s_j \in Ts\,, \tag{5.1}$$

where s_i (resp. s_j) is an object in the training set Tr (resp. test set Ts). Eq. 5.1 must hold; its violation inflates the results by including future knowledge in the classifier (Section 5.4.2).

C2) Temporal gw/mw windows consistency. In every test slot of size Δ, all test objects must be from the same time window:

$$t_i^{min} \leq time(s_k) \leq t_i^{max}, \quad \forall s_k \text{ in time slot } [t_i, t_i + \Delta)\,, \tag{5.2}$$

where $t_i^{min} = \min_k time(s_k)$ and $t_i^{max} = \max_k time(s_k)$. The same should hold for the training: although violating Eq. 5.2 in the training data does not bias the evaluation, it may affect the sensitivity of the classifier to unrelated artifacts. Eq. 5.2 has been violated in the past when goodware and malware have been collected from different time windows (*e.g.,* ALG2 [187], re-evaluated in Section 5.6)—if violated, the results are biased because the classifier may learn and test on artifactual behaviors that, for example, distinguish goodware from malware just by their different API versions.

...for each test period, objects should be from the same time window...

C3) Realistic malware-to-goodware ratio in testing. Let us define φ as the average percentage of malware in the training data, and δ as the average percentage of malware in the test data. Let $\hat{\sigma}$ be the estimated percentage of malware in the wild. To have a realistic evaluation, the average percentage of malware in the testing (δ) must be as close as possible to the percentage of malware in the wild ($\hat{\sigma}$), so that:

$$\delta \simeq \hat{\sigma}\,. \tag{5.3}$$

For example, we have estimated that in the Android scenario goodware is predominant over malware, with $\hat{\sigma} \approx 0.10$ (Section 5.3.2). If C3 is violated by overestimating the percentage of malware, the results are positively inflated (Section 5.4.3). We highlight that, although the test distribution δ cannot be changed (in order to get realistic results), the percentage of malware in the training φ may be tuned (Section 5.5.3).

...and the test set should approximate the expected base rate of the positive class in the wild.

5.5.2 *Time-aware Performance Metrics*

We introduce a time-aware performance metric that allows for the comparison of different classifiers while considering time decay. Let Θ be a classifier trained on *Tr*; we capture the performance of Θ for each time frame $[t_i, t_i + \Delta)$ of the test set *Ts* (*e.g.,* each month). We identify two options to represent per-month performance:

We propose AUT, a time-aware metric for comparing different methods...

- **Point estimates** (*pnt*): The value plotted on the *Y*-axis for $x_k = k\Delta$ (where k is the test slot number) computes the performance metric (*e.g.,* F_1-Score) only based on predictions $\hat{y}_i$ of Θ and true labels y_i in the interval $[W + (k-1)\Delta, W + k\Delta)$.
- **Cumulative estimates** (*cml*): The value plotted on the *Y*-axis for $x_k = k\Delta$ (where k is the test slot number) computes the performance metric (*e.g.,* F_1-Score) only based on predictions $\hat{y}_i$ of Θ and true labels y_i in the cumulative interval $[W, W + k\Delta)$.

Point estimates are always to be preferred to represent the real performance of an algorithm. The cumulative estimates can be used

to highlight a smoothed trend and to show overall performance up to a certain point, but can be misleading if reported on their own if objects are too sparsely distributed in some test slots Δ. Hence, we report primarily point estimates in the remainder of the chapter (*e.g.*, in Section 5.6), while an example of cumulative estimate plots is reported in Figure 5.6(b).

...which captures the curve of performance decay over time.

To facilitate the comparison of different time decay plots, we define a new metric, *Area Under Time* (**AUT**), the area under the performance curve over time. Formally, based on the trapezoidal rule (as in AUC [40]), AUT is defined as follows:

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(x_{k+1}) + f(x_k)]}{2}, \tag{5.4}$$

where: $f(x_k)$ is the value of the point estimate of the performance metric f (*e.g.*, F_1) evaluated at point $x_k := (W + k\Delta)$; N is the number of test slots, and $1/(N-1)$ is a normalization factor so that AUT $\in [0, 1]$. The perfect classifier with robustness to time decay in the time window S has AUT $= 1$. By default, AUT is computed as the area under point estimates, as they capture the trend of the classifier over time more closely; if the AUT is computed on cumulative estimates, it should be explicitly marked as AUT_{cml}. As an example, $\text{AUT}(F_1, 12m)$ is the point estimate of F_1-Score considering time decay for a period of 12 months, with a 1-month interval. We highlight that the simplicity of computing the AUT should be seen as a benefit rather than a drawback; it is a simple yet effective metric that captures the performance of a classifier with respect to time decay, de-facto promoting a fair comparison across different approaches.

> **AUT**(f,N) is a metric that allows us to evaluate performance f of a malware classifier against time decay over N time units in realistic experimental settings—obtained by enforcing C1, C2, and C3 (Section 5.5.1). The next sections leverage AUT for tuning classifiers and comparing different solutions (Section 5.6).

5.5.3 *Tuning Training Ratio*

We propose a novel algorithm that allows for the adjustment of the training ratio φ when the dataset is imbalanced, in order to optimize a user-specified performance metric (F_1, Precision, or Recall) on the minority class, subject to a maximum tolerated error, while aiming to reduce time decay. The high-level intuition of the impact of changing φ is described in Section 5.4.3. We also observe that ML literature has shown ROC curves to be misleading on highly imbalanced datasets [75, 126]. Choosing different thresholds on ROC curves *shifts* the decision boundary, but (as seen in the motivating example of Figure 5.3) re-training with different ratios φ (as in our

After examining the impact of altering the malware rate in the training set...

Algorithm 3: Tuning φ.

Input: Training dataset Tr
Parameters: Learning rate μ, target performance $\mathbb{P} \in \{F_1, Pr, Rec\}$, max error rate E_{max}
Output: $\varphi^*_{\mathbb{P}}$, optimal percentage of mw to use in training to achieve the best target performance $\mathbb{P}$ subject to $E < E_{max}$.

```
Split the training set Tr into two subsets: actual training (ProperTr) and validation set (Val), while enforcing C1, C2, C3
  (Section 5.5.1), also implying δ = δ̂
Divide Val into N non-overlapped subsets, each corresponding to a time-slot Δ, so that Val_array = [V_0, V_1, ..., V_N]
Train a classifier Θ on ProperTr
P* ← AUT(ℙ,N) on Val_array with Θ
φ*_ℙ = φ̂
for (φ = φ̂; φ ≤ 0.5; φ = φ + μ) do
    Downsample gw in ProperTr so that percentage of mw is φ
    Train the classifier Θ_φ on ProperTr with φ mw
    performance P_φ ← AUT(ℙ, N) on Val_array with Θ_φ
    error E_φ ← Error rate on Val_array with Θ_φ
    if (P_φ > P*) and (E_φ ≤ E_max) then
        P* ← P_φ
        φ*_ℙ ← φ
return φ*_ℙ;
```

algorithm) also changes the *shape* of the decision boundary, better representing the minority class.

Our tuning algorithm is inspired by one proposed by Weiss and Provost [309]; they propose a progressive sampling of training objects to collect a dataset that improves AUC performance of the minority class in an imbalanced dataset. However, they did not take temporal constraints into account (Section 5.4.2), and heuristically optimize only AUC. Conversely, we enforce C1, C2, C3 (Section 5.5.1), and rely on AUT to achieve three possible targets for the malware class: higher F_1-Score, higher Precision, or higher Recall. Also, we assume that the user already has a training dataset Tr and wants to use as many objects from it as possible, while still achieving a good performance trade-off; for this purpose, we perform a *progressive subsampling* of the goodware class.

...we devise an algorithm to find the optimal rate...

Algorithm 3 formally presents our methodology for tuning the parameter φ to find the value $\varphi^*_{\mathbb{P}}$ that optimizes $\mathbb{P}$ subject to a maximum error rate E_{max}. The algorithm aims to solve the following optimization problem:

$$\text{maximize}_{\varphi}\{\mathbb{P}\} \quad \text{subject to: } E \leq E_{max}\,, \tag{5.5}$$

where $\mathbb{P}$ is the target performance: the F_1-Score (F_1), Precision (Pr) or Recall (Rec) of the malware class; E_{max} is the maximum tolerated error; depending on the target $\mathbb{P}$, the error rate E has a different formulation:

- if $\mathbb{P} = F_1 \rightarrow E = 1 - \text{Acc} = {}^{(FP+FN)}/_{(TP+TN+FP+FN)}$
- if $\mathbb{P} = Rec \rightarrow E = FPR = {}^{FP}/_{(TN+FP)}$
- if $\mathbb{P} = Pr \rightarrow E = FNR = {}^{FN}/_{(TP+FN)}$

Each of these definitions of E is targeted to limit the error induced by the specific performance—if we want to maximize F_1 for the malware class, we need to limit both FPs and FNs; if $\mathbb{P} = Pr$, we increase FNs, so we constrain FNR.

Algorithm 3 consists of two phases: *initialization* (lines 1–5) and *grid search* of $\varphi^*_{\mathbb{P}}$ (lines 6–14). In the initialization phase, the training set *Tr* is split into a proper training set *ProperTr* and a validation set *Val*; this is split according to the space-time evaluation constraints in Section 5.5.1, so that all the objects in *ProperTr* are temporally anterior to *Val*, and the malware percentage δ in *Val* is equal to $\hat{\sigma}$, the in-the-wild malware percentage. The maximum performance observed P^* and the optimal training ratio $\varphi^*_{\mathbb{P}}$ are initialized by assuming the estimated in-the-wild malware ratio $\hat{\sigma}$ for training; in Android, $\hat{\sigma} \approx 10\%$ (see Section 5.3.2).

...which uses grid search to optimize a given performance metric.

The grid-search phase iterates over different values of φ, with a learning rate μ (*e.g.*, $\mu = 0.05$), and keeps as $\varphi^*_{\mathbb{P}}$ the value leading to the best performance, subject to the error constraint. To reduce the chance of discarding high-quality points while downsampling goodware, we prioritize the most uncertain points (*e.g.*, points close to the decision boundary in an SVM) [250]. The constraint on line 6 ($\hat{\sigma} \leq \varphi \leq 0.5$) is to ensure that one does not under-represent the minority class (if $\varphi < \hat{\sigma}$) and that one does not let it become the majority class (if $\varphi > 0.5$); also, from Section 5.4.3 it is clear that if $\varphi > 0.5$, then the error rate becomes too high for the goodware class. Finally, the grid-search explores multiple values of φ and stores the best ones. To capture time-aware performance, we rely on AUT (Section 5.5.2), and the error rate is computed according to the target $\mathbb{P}$ (see above). Tuning examples are in Section 5.6.

5.5.4 *Implementation of the* TESSERACT *Library*

We develop TESSERACT, an open-source Python framework that enforces constraints C1, C2, and C3 (Section 5.5.1), computes AUT (Section 5.5.2), and can train a classifier with our tuning algorithm (Section 5.5.3). TESSERACT operates as a traditional Python ML library but, in addition to features matrix X and labels y, it also takes as input the timestamp array t containing dates for each object.

To promote the adoption of our framework we release Tesseract...

TESSERACT is designed to integrate easily with common workflows, in particular, the API design of TESSERACT is heavily inspired by and fully compatible with the popular machine learning libraries SCIKIT-LEARN and TensorFlow's KERAS API. As a result, many of conventions and concepts in TESSERACT should be familiar to users of those libraries.

The goal of TESSERACT is to ensure a fair, time-aware evaluation of security classifiers. To achieve this, TESSERACT will enforce proper temporal and spatial constraints to prevent results becoming affected by experimental bias. As classifiers grow in complexity by combining multiple machine learning techniques, it becomes increasingly likely that these constraints will be violated. TESSERACT aims to reduce the burden on the system designer by keeping track of these properties at each stage of the experiment pipeline.

Evaluation Workflow. TESSERACT divides the workflow into stages (Figure 5.5). Firstly, the dataset is ordered chronologically and divided into a single training set and multiple test sets. Next, execution enters the *time-aware evaluation cycle* where each iteration of the cycle processes the subsequent set of test objects. The evaluation cycle is composed of multiple stages centered around the standard "training" and "prediction" procedures. The stage preceding training enables adjustments to be made to the training set while the later stages allow for policies for reacting to the results before the cycle repeats. Finally, once all test objects have been processed, the complete results are consolidated and presented.

...a modular Python library for performing evaluations free of spatio-temporal bias.

Modular Design. TESSERACT is composed in a modular fashion, to reflect the different stages of the evaluation cycle. Different phases of the cycle are represented by subclasses of `Stage`, which can themselves be subclassed to implement specific learning strategies. Instances of these subclasses can then be injected into the function `fit_predict_update()` which will activate them appropriately throughout the evaluation or deactivate them according to a given `schedule` (a boolean array) attached to the superclass. Alternatively, any component from the framework can be appropriately selected and used in conjunction with other libraries or methodologies. We hope the ease of writing plugins for different stages will encourage further experimentation with novel drift mitigation strategies. The following paragraphs further explore the core stages of TESSERACT's workflow.

Temporal Awareness. While a single training or test example is typically represented as a feature set X and an output variable, or ground truth, y, TESSERACT also expects a timestamp t. This allows TESSERACT to enforce temporal constraints when partitioning the dataset; e.g., for training, validation or testing. TESSERACT partitions test sets further into testing *periods*. Each period contains test objects covering a particular timespan specified in days, weeks, months, quarters or years.

Tesseract makes it easy to temporally partition data...

Time-aware operations are implemented in `temporal.py` which handles the various corner cases and complications that occur when working with time deltas. A notable function from the module is `time_aware_train_test_split()` that performs dataset partitioning given a time period length, granularity and an optional start date, such that all test periods are processed in chronological order and all are temporally posterior to the training set.

Pre-train Stages. Before training the classifier it can be beneficial to make adjustments to the training set. For example, altering the class balance of the training set can be used to tune a classifier in order to make it more or less receptive to a particular class. This is especially useful in many security applications where the class of interest—often malicious—is also the minority class.

...and fix the class ratio of the training and test sets.

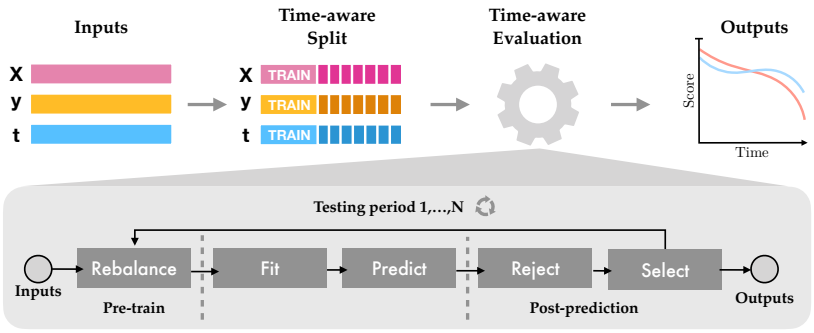


Figure 5.5: TESSERACT workflow and the evaluation cycle.

The module `spatial.py` contains `downsample_set()` to reduce the majority class until the desired class balance is achieved, as well as an implementation of our tuning algorithm (Section 5.5.3) in `search_optimal_train_ratio()`. Custom methods for these adjustments can be injected by subclassing the `Rebalancer` class and overriding its `alter()` method which will then be invoked before training on each cycle iteration.

The function `downsample_set()` can also be used to ensure the class balance of each testing period is *realistic*. For this, `spatial.py` also includes `assert_class_distribution()` to enforce C3 within some tolerance bound.

A flexible evaluation cycle with plugin support...

Train and Predict Stages. During training, the classifier estimates the relationship between the features and the output variables. By default, TESSERACT invokes the `fit` function on the given model, however, custom algorithms can be injected into the fitting stage to aid interoperability or for experimentation.

In the next stage, the classifier attempts to predict correct classes for the test objects in the current period. TESSERACT will search for typical classification functions—prioritizing those which output raw scores (*e.g.*, distance from hyperplane) and thus are more flexible for calculating metrics. However, custom decision functions can also be passed to the framework to override the default behavior.

Post-classification Stages. In Section 5.7.1 we discuss techniques for delaying time decay: incremental retraining, classification with rejection [139], and active learning [250]. In these approaches, examples may be rejected or manually labeled. In most cases, there is an associated *quarantine cost*, as the associated manual inspection consumes time and resources. TESSERACT tracks quarantine costs as these are important for comparing delay strategies and can also signal the onset of *concept drift* and the aging of the model [139].

...makes it easy to extend Tesseract to test novel strategies for mitigating performance decay.

The modules `rejection.py` and `selection.py` provide hooks for novel query and reject strategies to be easily plugged into the evaluation cycle. Policies for rejection, while optional, can be implemented by subclassing the `Rejector` stage class and overriding the `reject()` method.

Other delay strategies are typically activated after rejection and before the rebalancing and retraining of the next evaluation cycle. Active learning techniques can be implemented in TESSERACT by subclassing the `Selector` stage and overriding the `query()` method. Similarly to `Rejector` objects, all `Selector` stages will keep track of costs they incur. TESSERACT includes some useful implementations for this stage, for example `UncertaintySamplingSelector`. Incremental retraining can be simulated employed by using the `FullRetrainingSelector`.

Metrics and Output. TESSERACT maintains a set of metrics calculated during each iteration of the evaluation cycle. These range from the total positive and negative objects to metrics such as Precision, Recall, and AUC. As TESSERACT aims to encourage comparable and reproducible evaluations, we include functions for visualizing classifier assessments and for measuring the classifier robustness over a given time period with respect to each metric.

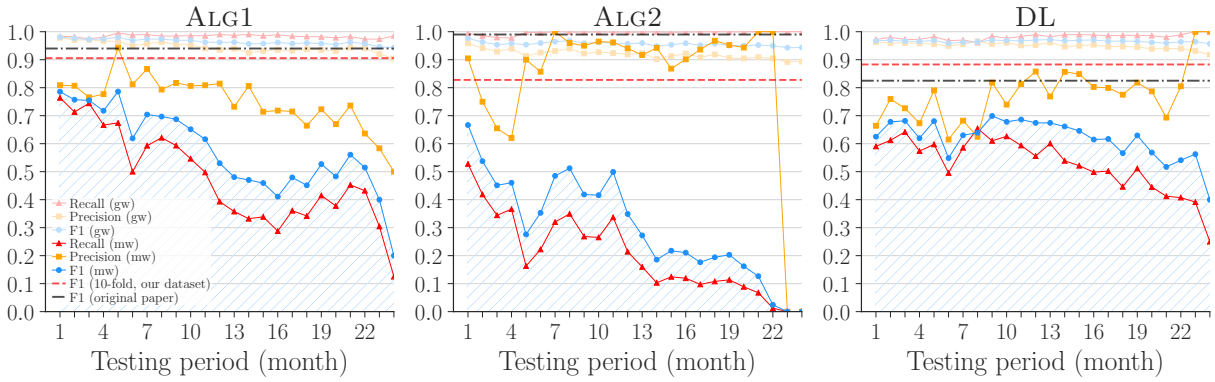
5.6 TESSERACT: REVEALING HIDDEN PERFORMANCE

Here, we show how our methodology can reveal hidden performance of ALG1 [19], ALG2 [187], and DL [118] (Section 5.3.1), and their robustness to time decay.

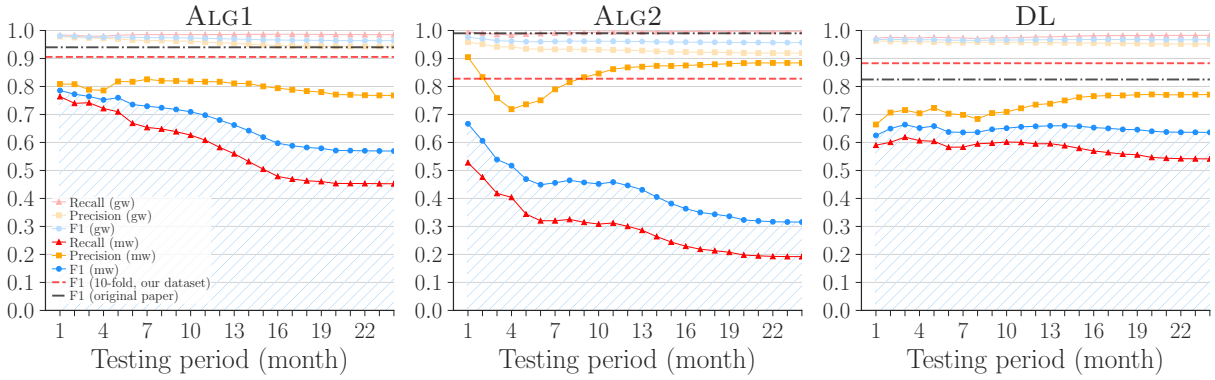
Figure 5.6 reports several performance metrics of the three algorithms as point estimates over time. The X-axis reports the testing slots in months, whereas the Y-axis reports different scores between 0 and 1. The areas highlighted in blue correspond to the AUT($F_1, 24m$). The black dash-dotted horizontal lines represent the best F_1-Score from the original papers [19, 187, 118], corresponding to results obtained with 10 hold-out random splits for ALG1, 10-fold CV for ALG2, and random split for DL; all these settings are analogous to k-fold from a temporal bias perspective, and violate both C1 and C2. The red dashed horizontal lines correspond to 10-fold F_1-Score obtained on our dataset, which satisfies C3.

A thorough evaluation of the reference algorithms reveals more realistic performances...

Differences in 10-fold F_1-Score. We discuss and motivate the differences between the horizontal lines representing original papers' best F_1-Score and replicated 10-fold F_1-Score. The 10-fold F_1-Score of ALG1 is close to the original paper [19]; the difference is likely related to the use of a different, more recent dataset. The 10-fold F_1-Score of ALG2 is much lower than the one in the paper. We verified that this is mostly caused by **violating C3**: the best F_1-Score reported in [187] is on a setting with 86% malware—hence, spatial bias increases even 10-fold F_1-Score of ALG2. Also **violating C2** tends to inflate the 10-fold performance as the classifier may learn artifacts. The 10-fold F_1-Score in DL is instead slightly higher than in the original paper [118]; this is likely related to a hyperparameter tuning in the original paper that optimized Accuracy (instead of



(a) Point estimates



(b) Cumulative estimates

Figure 5.6: Time decay of ALG1 [19], ALG2 [187], and DL [118]—with AUT$(F_1, 24m)$ of 0.58, 0.32, and 0.64, respectively. Training and test sets with 10% malware.

F_1-Score), which is known to be misleading in imbalanced datasets. From these results, we can observe that even if an analyst wants to estimate what the performance of the classifier would be in the *absence* of concept drift (*i.e.,* where objects coming from the same distribution of the training dataset are received by the classifier), they still need to enforce C2 and C3 while computing 10-fold CV to obtain valid results.

...with much weaker results once sources of bias are removed...

Violating C1 and C2. Removing the temporal bias reveals the real performance of each algorithm in the presence of concept drift. The AUT$(F_1, 24m)$ quantifies such performance: 0.58 for ALG1, 0.32 for ALG2 and 0.64 for DL. In all three scenarios, the AUT$(F_1, 24m)$ is lower than 10-fold F_1-Score as the latter violates constraint C1 and may violate C2 if the dataset classes are not evenly distributed across the timeline (Section 5.5).

Best performing algorithm. TESSERACT shows a counterintuitive result: the algorithm that is most robust to time decay and has the highest performance over the 2 years testing is the DL algorithm (after removing space-time bias), although for the first few months ALG1 outperforms DL. Given this outcome, one may prefer to use ALG1 for the first few months and then DL, if retraining is not possible (Section 5.7). We observe that this strongly contradicts

...and that DL *is most robust to time decay in the long term, despite reporting the worst performance originally [118].*

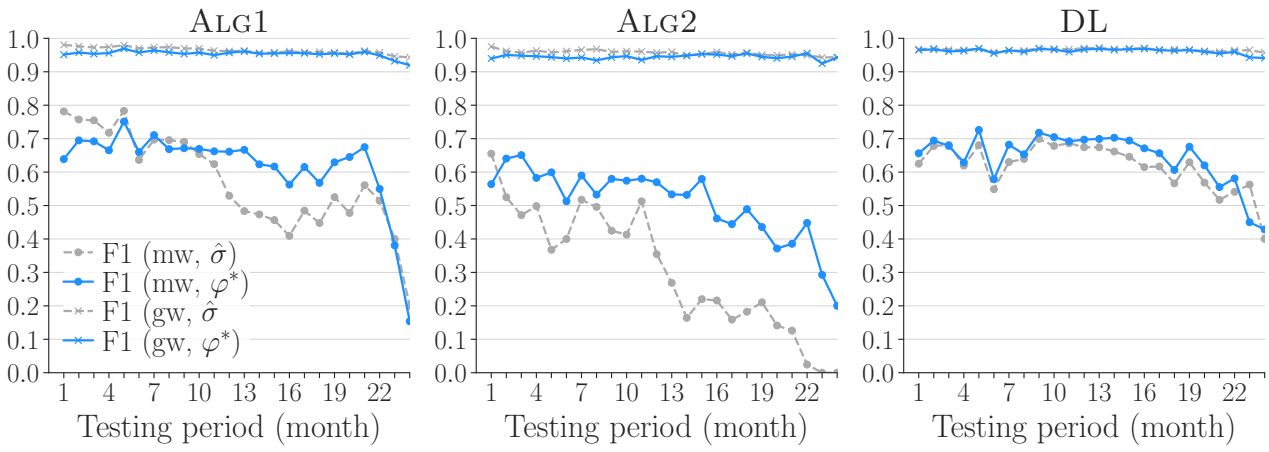


Figure 5.7: Improvement obtained by applying $\varphi^*_{F_1} = 25\%$ to both ALG1 and DL, and $\varphi^*_{F_1} = 50\%$ to ALG2. $\varphi^*_{F_1}$ values are obtained with Algorithm 3 on the training set (split 2:1 training to validation).

the performance obtained in the presence of temporal and spatial bias. In particular, if we only looked at the best F_1-Score reported in the original papers, ALG2 would have been the best algorithm (because spatial bias was present). After enforcing C3, the k-fold on our dataset would have suggested that DL and ALG1 have similar performance (because of temporal bias). After enforcing C1, C2 and C3, the AUT reveals that DL is actually the algorithm most robust to time decay.

Different robustness to time decay. Given a training dataset, the robustness of different ML models against performance decay over time depends on several factors. Although more in-depth evaluations would be required to understand the theoretical motivations behind the different robustness to time decay of the three algorithms in our setting, we hereby provide insights on possible reasons. The performance of ALG2 is the fastest to decay likely because its feature engineering [187] may be capturing relations in the training data that quickly become obsolete at test time to separate goodware from malware. Although ALG1 and DL take as input the same feature space, the higher robustness to time decay of DL is likely related to feature representation in the *latent feature space* automatically identified by deep learning [118], which appears to be more robust to time decay in this specific setting. Recent results have also shown that linear SVM tends to overemphasize a few important features [191]—which are the few most effective on the training data, but may become obsolete over time. We remark that we are not claiming that deep learning is necessarily more robust to time decay than traditional ML algorithms. Instead, we demonstrate how, in this specific setting, TESSERACT allowed us to highlight higher robustness of DL [118] against time decay; however, the prices to pay to use DL are lower explainability [235, 122] and higher training time [118].

Tuning algorithm. We now evaluate whether our tuning (Algorithm 3 in Section 5.5.3) improves robustness to time decay of a malware classifier for a given target performance. We first aim to maximize $\mathbb{P} = F_1$-Score of malware class, subject to E_{max}= 10%. After running Algorithm 3 on ALG1 [19], ALG2 [187] and DL, we find that $\varphi^*_{F_1} = 0.25$ for ALG1 and DL, and $\varphi^*_{F_1} = 0.5$ for ALG2. Figure 5.7 reports the improvement on the test performance of applying $\varphi^*_{F_1}$ to the full training set Tr of 1 year. We remark that the choice of $\varphi^*_{F_1}$ uses only training information (see Algorithm 3) and no test information is used—the optimal value is chosen from a 4-month validation set extracted from the 1 year of training data; this is to simulate a realistic deployment setting in which we have no a priori information about testing. Figure 5.7 shows that our approach for finding the best $\varphi^*_{F_1}$ improves the F_1-Score on malware at test time, at the cost of slightly reduced goodware performance. Table 5.3 shows details of how total FPs, total FNs, and AUT changed by training ALG1, ALG2, and DL with $\varphi^*_{F_1}$, φ^*_{Prec}, and φ^*_{Rec} instead of $\hat{\sigma}$. These training ratios have been computed subject to $E_{max} = 5\%$ for φ^*_{Rec}, $E_{max} = 10\%$ for $\varphi^*_{F_1}$, and $E_{max} = 15\%$ for φ^*_{Prec}; the difference in the maximum tolerated errors is motivated by the class imbalance in the dataset—which causes lower FPR and higher FNR values (see definitions in Section 5.5.3), as there is much more goodware than malware. As expected (Section 5.4.3), Table 5.3 shows that when training with $\varphi^*_{F_1}$ Precision decreases (FPs increase) but Recall increases (because FNs decrease), and the overall AUT increases slightly as a trade-off. A similar reasoning follows for the other performance targets. We observe that the AUT for Precision may slightly differ even with a similar number of total FPs—this is because AUT$(Pr, 24m)$ is sensitive to the time at which FPs occur; the same observation is valid for total FNs and AUT Recall. After tuning, the F_1 performance of ALG1 and DL become similar, although DL remains higher in terms of AUT. The tuning improves the AUT$(F_1, 24m)$ of DL only marginally, as DL is already robust to time decay even before tuning (Figure 5.6).

We perform more tuning with our class balance tuning algorithm...

...showing that it can further improve robustness against drift.

The next section focuses on the two classifiers less robust to time decay, ALG1 and ALG2, to evaluate with TESSERACT the performance-cost trade-offs of budget-constrained strategies for delaying time decay.

5.7 Delaying Time Decay

We have shown how enforcing constraints and computing AUT with TESSERACT can reveal the real performance of Android malware classifiers (Section 5.6). This *baseline AUT performance* (without retraining) allows users to evaluate the general robustness of an algorithm to time decay. A classifier may be retrained to update its model. However, *manual labeling* is costly (especially in the Android malware setting), and the ML community [250, 30] has worked

Algorithm	φ	FP	FN	AUT($\mathbb{P}$, 24m)		
				F_1	Pr	Rec
ALG1 [19]	10% ($\hat{\sigma}$)	965	3,851	0.58	0.75	0.48
	25% ($\varphi^*_{F_1}$)	2,156	2,815	0.62	0.65	0.61
	10% (φ^*_{Pr})	965	3,851	0.58	0.75	0.48
	50% (φ^*_{Rec})	3,728	1,793	0.64	0.58	0.74
ALG2 [187]	10% ($\hat{\sigma}$)	274	5,689	0.32	0.77	0.20
	50% ($\varphi^*_{F_1}$)	4,160	2,689	0.53	0.50	0.60
	10% (φ^*_{Pr})	274	5,689	0.32	0.77	0.20
	50% (φ^*_{Rec})	4,160	2,689	0.53	0.50	0.60
DL [118]	10% ($\hat{\sigma}$)	968	3,291	0.64	0.78	0.53
	25% ($\varphi^*_{F_1}$)	2,284	2,346	0.65	0.66	0.65
	10% (φ^*_{Pr})	968	3,291	0.64	0.78	0.53
	25% (φ^*_{Rec})	2,284	2,346	0.65	0.66	0.65

Table 5.3: Test AUT performance over 24 months, training with $\hat{\sigma}$, $\varphi^*_{F_1}$, φ^*_{Pr} and φ^*_{Rec}.

extensively on mitigation strategies—*e.g.,* to identify a limited number of *best* objects to label (active learning). While effective at postponing time decay, strategies like these can further complicate the fair evaluation and comparison of classifiers.

With a stable evaluation setup we can now compare different defenses...

In this section, we show how TESSERACT can be used to compare and evaluate the trade-offs of different budget-constrained strategies to delay time decay. Since DL has shown to be more robust to time decay (Section 5.6) than ALG1 and ALG2, in this section we focus our attention on these to show the performance-cost trade-offs of different drift mitigations.

5.7.1 *Delay Strategies*

We do not propose novel delay strategies, but instead focus on how TESSERACT allows for the comparison of some popular approaches to mitigating time decay. This shows researchers how to adopt TESSERACT for the fair comparison of different approaches when proposing novel solutions to delaying time decay under budget constraints. We now summarize the delay strategies we consider and show results on our dataset.

Incremental retraining. We first consider an approach that tends towards an "ideal" performance P^*: *all* test objects are periodically labeled manually, and the new knowledge introduced to the classifier via retraining. More formally, the performance of month m_i is determined from the predictions of a model Θ trained on: $Tr \cup \{m_0, m_1, ..., m_{i-1}\}$, where $\{m_0, m_1, ..., m_{i-1}\}$ are test objects, which are manually labeled. The dashed gray line represents the F_1-Score *without* incremental retraining (*i.e.,* stationary training). Although incremental retraining generally achieves close to optimal performance throughout the whole test period, it also incurs the

...such as incremental retraining...

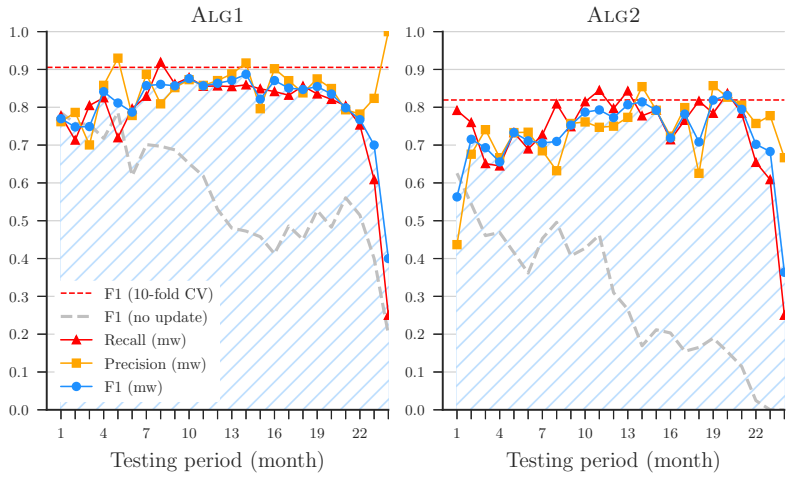


Figure 5.8: Delaying time decay: performance with incremental retraining.

highest labeling cost L and is often infeasible in realistic deployments. Even assuming a reliance on VirusTotal, there is still an API usage cost associated with higher query rates and the approach may be ill-suited to other security domains. Figure 5.8 shows the performance of ALG1 and ALG2 with monthly retraining.

...active learning...

Active learning. Active Learning (AL) is an area of machine learning that studies *query strategies*: strategies for selecting a subset of test objects (with unknown labels) that, if manually labeled and included in the training set, should be the most valuable for updating the classification model [250].

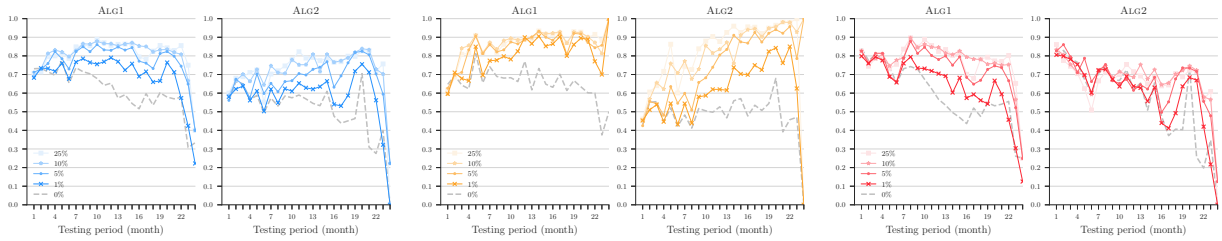
We evaluate the impact of one of the most popular active learning strategies: *uncertainty sampling* [250, 194]. This query strategy selects the points the classifier is least certain about, and uses them for retraining—the intuition is that the most uncertain elements are the ones that may be indicative of concept drift, and new, correct knowledge about them may better inform the decision boundaries. This intuition is clear for an algorithm such as linear SVM where the least certain points are those closest to the decision boundary. In a linear SVM the slope of the decision boundary greatly depends on the points that are closest to it, the *support vectors* [40]; all points further away are classified with higher confidence, hence have limited effect on the slope of the hyperplane.

More formally, in binary classification uncertainty sampling gives a score x^*_{LC} (where LC stands for *Least Confident*) to each sample [250]; this score is defined as follows:[2]

$$x^*_{LC} := \text{argmax}_x \{ 1 - P_\Theta(\hat{y}|x) \}, \tag{5.6}$$

where $\hat{y} := \text{argmax}_y P_\Theta(y|x)$ is the class label with the highest posterior probability according to classifier Θ. In a binary classification task, the maximum uncertainty for an object is achieved when its prediction probability is equal to 0.5 for both classes (*i.e.*, equal probability of being goodware or malware). The test objects are sorted by descending order of uncertainty x^*_{LC}, and the top-n most

[2] In multi-class classification, there is a query strategy based on the entropy of the prediction scores array; in binary classification, the entropy-based query strategy is proven to be equivalent to the LC strategy [250].



(a) F_1-Score (b) Precision (c) Recall

Figure 5.9: Delaying time decay: performance with active learning based on uncertainty sampling.

uncertain are selected to be labeled for retraining the classifier.

Depending on the percentage of manually labeled points, each scenario corresponds to a different labeling cost L. The labeling cost L is known a priori since it is user specified according to their available resources.

We apply active learning with uncertainty sampling in a time-aware scenario, and choose a percentage of objects to retrain per month. Figure 5.9 reports the results for different percentages of objects labeled per month. We observe that even with 1% AL, the performance already improves significantly.

Classification with rejection. Malware evolves rapidly over time, so if the classifier is not up to date, the decision region may no longer be representative of new objects. Another approach, orthogonal to active learning, is to include a *reject option* as a possible classifier outcome [107, 139]. This discards the most uncertain predictions to a *quarantine* area for manual inspection at a future date. At the cost of rejecting some objects, the overall performance of the classifier (on the remaining objects) increases. The intuition is that in this way only high confidence decisions are taken into account. Again, although performance P improves, there is a quarantine cost Q associated with it; in this case, unlike active learning, the cost is not known a priori because, in traditional classification with rejection, a threshold on the classifier confidence is applied [107, 139]. While we assess a simple method here, Chapter 6 examines classification with rejection methods in greater detail.

...and classification with rejection.

Figure 5.10 reports the performance of ALG1 and ALG2 after applying a reject option based on Jordaney et al. [139]. In particular, we use the third quartile of probabilities of incorrect predictions as the rejection threshold [139]. The gray histograms in the background report the number of rejected objects per month. The second year of testing has more rejected objects for both ALG1 and ALG2, although ALG2 overall rejects more objects.

5.7.2 *Analysis of Delay Methods*

To quantify performance-cost trade-offs of methods to delay time decay without changing the algorithm, we characterize the follow-

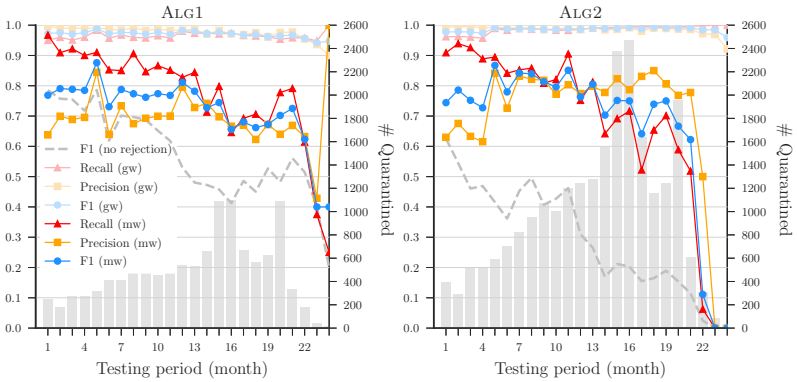


Figure 5.10: Delaying time decay: performance with classification with rejection.

ing three elements: **Performance** (P), the performance measured in terms of AUT to capture robustness against time decay (Section 5.5.2); **Labeling Cost** (L), the number of test objects (if any) that must be labeled—the labeling must occur periodically (*e.g.*, every month), and is particularly costly in the malware domain as manual inspection requires many resources (infrastructure, time, expertise, etc)—for example, Miller et al. [194] estimate an average company could manually label 80 objects per day; **Quarantine Cost** (Q), the number of objects (if any) rejected by the classifier—these must be manually verified, so there is a cost for leaving them in quarantine.

Table 5.4, utilizing AUT(F_1,24m) while enforcing our constraints, summarizes labeling cost L, quarantine cost Q, and two performance columns P, corresponding to training with $\hat{\sigma}$ and $\varphi^*_{F_1}$ (Section 5.5.3), respectively. In each row, we highlight in orange and purple cells the column with the highest AUT for ALG1 and ALG2, respectively. Table 5.4 allows us to: (i) examine the effectiveness of the training ratios $\varphi^*_{F_1}$ and $\hat{\sigma}$; (ii) analyze the AUT performance improvement and the costs for delaying time decay; (iii) compare the performance of ALG1 and ALG2 in different settings.

We can clearly compare the cost and performance tradeoffs...

First, let us compare $\varphi^*_{F_1}$ with $\hat{\sigma}$. The first row of Table 5.4 represents the scenario in which the model is trained only once at the beginning—the scenario for which we originally designed Algorithm 3 (Section 5.5.3 and Figure 5.7). Without methods to delay time decay, $\varphi^*_{F_1}$ achieves better performance than $\hat{\sigma}$ for both ALG1 and ALG2 at no cost. In all other configurations, we observe that training $\varphi = \varphi^*_{F_1}$ always improves performance for ALG2, whereas for ALG1 it is slightly advantageous in most cases except for rejection and AL 1%—in general, the performance of ALG1 trained with $\varphi^*_{F_1}$ and $\hat{\sigma}$ is consistently close. The intuition is that $\varphi^*_{F_1}$ and $\hat{\sigma}$ are also close for ALG1: when applying the AL strategy, we re-apply Algorithm 3 at each step and find that the average $\varphi^*_{F_1} \approx 15\%$ for ALG1, which is close to 10% (*i.e.*, $\hat{\sigma}$). On the other hand, for ALG2 the average $\varphi^*_{F_1} \approx 50\%$, which is far from $\hat{\sigma}$ and improves all results significantly. We can conclude that our tuning algorithm is most effective when it finds a $\varphi^*_{\mathbb{P}}$ that differs from the estimated $\hat{\sigma}$.

Then, we analyze the performance improvement and related cost of using delay methods. The improvement in F_1-Score granted

	Costs				Performance			
	L		Q		P : AUT(F_1, 24m)			
					$\varphi = \hat{\sigma}$		$\varphi = \varphi^*_{F_1}$	
Method	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2
No update	0	0	0	0	0.577	0.317	0.622	0.527
Rejection ($\hat{\sigma}$)	0	0	10,283	3,595	0.717	0.280	–	–
Rejection ($\varphi^*_{F_1}$)	0	0	10,576	24,390	–	–	0.704	0.683
AL: 1%	709	709	0	0	0.708	0.456	0.703	0.589
AL: 2.5%	1,788	1,788	0	0	0.738	0.509	0.758	0.667
AL: 5%	3,589	3,589	0	0	0.782	0.615	0.784	0.680
AL: 7.5%	5,387	5,387	0	0	0.793	0.641	0.801	0.714
AL: 10%	7,189	7,189	0	0	0.796	0.656	0.802	0.732
AL: 25%	17,989	17,989	0	0	0.821	0.674	0.823	0.732
AL: 50%	35,988	35,988	0	0	0.817	0.679	0.828	0.741
Inc. retrain	71,988	71,988	0	0	0.818	0.679	0.830	0.736

Table 5.4: Performance-cost comparison of delay methods.

by our algorithm comes at no labeling or quarantine cost. We can observe that one can improve the in-the-wild performance of the algorithms at some cost L or Q. It is important to observe that objects discarded or to be labeled are not necessarily malware; they are just the objects most uncertain according to the algorithm, which the classifier may have likely misclassified. The labeling costs L for ALG1 and ALG2 are identical (same dataset); in AL, the percentage of retrained objects is user-specified and fixed.

...between different algorithms and mitigation strategies.

Finally, Table 5.4 shows that ALG1 consistently outperforms ALG2 on F_1 for all performance-cost trade-offs. This confirms the trend seen in the realistic settings of Table 5.2.

This section shows that TESSERACT is helpful to both researchers and industrial practitioners. Practitioners need to estimate the performance of a classifier in the wild, compare different algorithms, and determine resources required for L and Q. For researchers, it is useful to understand how to reduce costs L and Q while improving performance P through comparable, unbiased evaluations. The problem is challenging, but we hope that releasing TESSERACT's code fosters further research and widespread adoption.

5.8 BEYOND ANDROID MALWARE

Although we applied TESSERACT to the Android domain, our methodology is general and can be immediately applied to any machine learning-driven security domain to achieve an evaluation without spatio-temporal bias. However, our methodology does require some domain-specific parameters in order to reflect realistic conditions, namely access to large timestamped datasets, knowledge of realistic class ratios, and code or sufficient details to reproduce baselines. This is not a weakness of our work, but rather an expected requirement.

We identify numerous constraint violations across 36 papers from the community...

As further evidence to motivate the adoption of realistic evaluation settings, we conduct a survey of security detection papers across a 10 year period, from 2011 to 2018. The study analyzes the

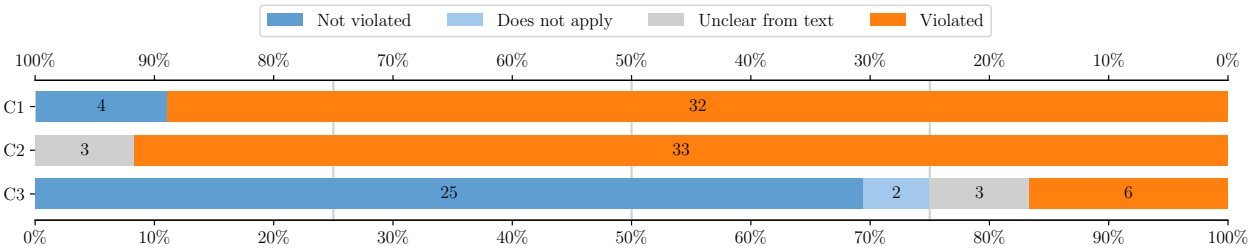


Figure 5.11: Stacked bar chart showing constraints violated by each of the 36 papers analyzed. The width of each bar shows the proportion of papers that violated each constraint while the number at the center of each bar shows the cardinality of each group.

evaluation setup of 36 papers, including 15 papers published at ACM CCS, IEEE S&P, USENIX Security, and NDSS—the top-4 conferences for security-related research in our community. Figure 5.12 shows a breakdown of the papers by year of publication.

The survey covers ALG1 [19], ALG2 [187], DL [119], and additional papers from the Android domain [109, 277, 276, 55, 74, 246, 329]; other malware domains such as Windows [245, 71, 281, 188, 46, 286, 254], PDF [271, 54, 264, 164], malicious JavaScript [68, 70, 236, 128], and ActionScript [208]; and other tasks such as malicious URL detection [274, 45], social media abuse prevention [42, 269], traffic analysis [225], vulnerability discovery [172], game bot detection [166], bulletproof hosting [9], protocol tunneling [29], and online scam detection [149].

The aggregated results are shown in Figure 5.11. A bar's color indicates whether a constraint (C1, C2, or C3) was violated in the evaluation, and the width shows the proportion of papers under that categorization. The number of papers affected is marked at the center of each bar.

The results show that a great majority of previous work has been affected by temporal bias, with 89% and 92% of papers violating C1 and C2, respectively. In particular, it could not be confirmed for any of the papers that C2 was not violated, due to a lack of clarity regarding dataset composition in three of the works.

While spatial bias is less prevalent, up to a quarter of papers surveyed may have suffered from the issue, and we were able to positively identify C3 violations in 17% of the papers.

These results indicate that experimental bias has been endemic in ML-based security evaluations, affecting many detection problems tackled by the security community. All but two papers violate at least one constraint, and for those two papers it was not clear from the text if C2 was or was not violated.

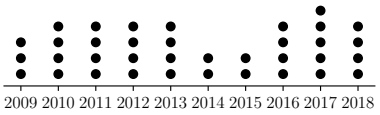


Figure 5.12: Distribution of papers per year for the 36 papers in this survey.

...demonstrating that spatio-temporal bias is not confined to our Android malware case studies.

Remark. All of the papers included in this survey provide valuable insights, and the presence of bias does not necessarily invalidate their contributions. Our aim is not to blame specific researchers, but to help raise awareness and provide solutions for these experimental issues.

5.9 TESSERACT OPERATION AND LIMITATIONS

We now discuss guidelines, our assumptions, and how we address limitations of our work.

Actionable points on Tesseract. It is relevant to discuss how both researchers and practitioners can benefit from TESSERACT and our findings. A *baseline AUT performance* (without classifier retraining) allows users to evaluate the general robustness of an algorithm to performance decay (Section 5.5.2). We demonstrate how TESSERACT can reveal true performance and provide counterintuitive results (Section 5.6). Robustness over extended time periods is practically relevant for deployment scenarios without the financial or computational resources to label and retrain often. Even with retraining strategies (Section 5.7), classifiers may not perform consistently over time. Manual labeling is costly, and the ML community has worked on mitigation strategies to identify a limited number of *best* objects to label (*e.g.*, active learning [250]). TESSERACT takes care of removing spatio-temporal bias from evaluations, so that researchers can focus on the proposal of more robust algorithms (Section 5.7). In this context, TESSERACT allows for the creation of comparable baselines for algorithms in a time-aware setting. Moreover, TESSERACT can be used with different time granularities, provided each period has a significant number of samples. For example, if researchers are interested in increasing robustness to decay for the upcoming 3 months, they can use TESSERACT to produce bias-free comparisons of their approach with prior research, while considering time decay.

Tesseract is flexible and aims to be useful to both academic researchers and industry practitioners.

Choosing time granularity (Δ). Choosing the length of the time slots (*i.e.*, time granularity) largely depends on the sparseness of the available dataset: in general, the granularity should be chosen to be as small as possible, while containing a statistically significant number of samples—as a rule of thumb, we keep the buckets large enough to have at least 1000 objects, which in our case leads to a monthly granularity. If there are restrictions on the number of time slots that can be considered (perhaps due to limited processing power), a coarser granularity can be used; however if the granularity becomes too large then the true trend might not be captured.

Different domains, datasets, and design goals will have their own individual requirements.

Identifying domain-specific malicious prevalence $\hat{\sigma}$. In the Android landscape, we assume that $\hat{\sigma}$ is around 10% (Section 5.3.2). Correctly estimating the malware percentage in the testing dataset is a challenging task and we encourage further representative measurement studies [173, 294] and data sharing to obtain realistic experimental settings.

Label accuracy. We assume goodware and malware labels in the dataset are correct (Section 5.3.3). Miller et al. [194] found that

AVs sometimes change their outcome over time: some goodware may eventually be tagged as malware. However, they also found that VirusTotal detections stabilize after one year; since we are using observations up to Dec 2016, we consider VirusTotal's labels as reliable. In the future, we may integrate approaches for *noisy oracles* [82], which assume some observations are mislabeled.

Timestamps accuracy. It is important to consider that some timestamps in a public dataset could be incorrect or invalid. In this chapter, we rely on the public AndroZoo dataset maintained at the University of Luxembourg, and we rely on the `dex_date` attribute as the approximation of an observation timestamp, as recommended by the dataset creators [8]. We further verified the reliability of the `dex_date` attribute by re-downloading VirusTotal [115] reports for 25K apps[3] and verifying that the `first_seen` attribute always matched the `dex_date` within our time span. In general, we recommend performing some sanitization of a timestamped dataset before performing any analysis on it: if multiple timestamps are available for each object, consider the most reliable timestamp you have access to (*e.g.,* the timestamp recommended by the dataset creators, or the VirusTotal's `first_seen` attribute) and discard objects with "impossible" timestamps (*e.g.,* with dates which are either too old or in the future), which may be caused by incorrect parsing or invalid values of some timestamps. To improve trustworthiness of the timestamps, one could verify whether a given object contains time inconsistencies or features not yet available when the app was released [170]. We encourage the community to promptly notify dataset maintainers of any date inconsistencies.

[3] We download only 25K VT reports (corresponding to about 20% of our dataset) due to the limitations of our VirusTotal API quota.

The ultimate reliability of different timestamp sources remains an open question.

Resilience of malware classifiers. In our study, we analyze three recent high-profile classifiers. One could argue that other classifiers may show consistently high performance even with space-time bias eliminated. And this should indeed be the goal of research on malware classification. TESSERACT provides a mechanism for an unbiased evaluation that we hope will support this kind of work.

Adversarial ML. Adversarial ML focuses on perturbing training or test inputs to compel a classifier to make incorrect predictions [34]. As described in Chapter 3, in security we can view concept drift as an emergent phenomenon largely driven by adversarial activity. While defenses against adversarial examples remain an open problem, the experimental bias we describe in this chapter—endemic in Android malware classification—must be addressed prior to realistic evaluations of adversarial mitigations.

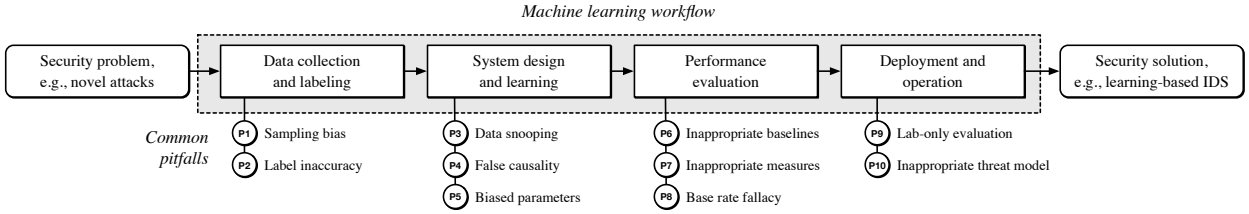


Figure 5.13: Common pitfalls of machine learning in computer security.

5.10 Other Sources of Experimental Bias in ML

Aside from the spatio-temporal bias we have discussed in depth, there are other sources of bias that can affect machine learning-based experiments. For completeness, we here describe 10 *pitfalls* which have afflicted machine learning-based security research in the past decade. With a few exceptions, these biases can typically apply to all machine learning studies, although they often have specific implications when applied to the security domain.

Beyond spatio-temporal biases, evaluations can suffer from more general sources of bias...

Unlike the spatio-temporal bias, these pitfalls may not necessarily be caused by specific properties of the hostile environment or of the malicious class. However, it is necessary to address as many sources of bias as possible in order to have a stable and consistent evaluation design to assess security detection systems in realistic settings. For this reason, we offer insight into the 10 pitfalls here.

...affecting different stages of an ML pipeline.

5.10.1 *Pitfall Definitions and Recommendations*

We describe ten common pitfalls that occur frequently in security research. Although some of these pitfalls may seem obvious at first glance, they are rooted in subtle mistakes that are widespread in security research—even in papers presented at top conferences (see Section 5.10.3).

As visualized in Figure 5.13, the pitfalls can be divided into categories based on which phase of the machine learning pipeline they pertain to: *data collection and labeling* (P1 and P2), *system design and learning* (P3, P4, P5), *performance evaluation* (P6, P7, P8), and *deployment and operation* (P9 and P10).

For each pitfall, we provide a short description and discuss its impact on the security domain, as well as providing some recommendations for mitigation. For deeper analysis into the prevalence and impact of each pitfall, we refer the reader to Arp et al. [20].

P1) Sampling Bias. *The collected data does not sufficiently represent the true data distribution of the underlying security problem* [67, 1, 62].

Sampling bias occurs when the distribution of a dataset does not effectively represent that expected in the wild...

Description. With a few rare exceptions, researchers develop their learning-based approaches without exact knowledge of the true underlying distribution of the input space. Instead, they need to

rely on a dataset containing a fixed number of samples that aim to resemble the actual distribution. While it is inevitable that some bias exists in most cases, understanding the specific bias inherent to a particular problem is crucial to limit its impact in practice. Drawing meaningful conclusions from the training data becomes challenging, if the data does not effectively represent the input space or even follows a different distribution.

Security implications. Sampling bias is highly relevant to security, as the acquisition of data is particularly challenging and often requires using multiple sources of varying quality. As an example, for the collection of suitable datasets for Android malware detection only a few public sources exist from which to obtain such data [307, 8]. As a result, it is common practice to combine data from different sources, which can introduce severe biases.

Recommendations. In many security applications, sampling from the true distribution is extremely difficult, if not impossible, requiring alternative solutions. A reasonable strategy is to construct different estimates of the true distribution and analyze them individually. Other valid strategies include the extension of the dataset with synthetic data [*e.g.*, 56, 123, 313] or the use of methods from the field of transfer learning [see 209, 310, 330, 328]. The mixing of data from distinct or incompatible sources should be avoided, as it is a common cause of sampling bias. In any case, possible limitations of the used dataset should always be discussed, allowing other researchers to better understand the security implications and impact of the underlying sampling bias.

...but is hard to mitigate completely in security—though methods from data augmentation and transfer learning can help.

P2) Label Inaccuracy. *The ground-truth labels required for classification tasks are inaccurate, unstable, or erroneous, affecting the overall performance of a learning-based system* [176, 327].

Description. Many learning-based security systems are built for classification tasks. To train these systems, a ground-truth label is required for each observation. Unfortunately, the ground truth is rarely perfect and researchers must account for uncertainty and noise to prevent their models from suffering from inherent bias.

Label inaccuracy results from weak labeling heuristics which can also be affected by drift...

Security implications. For many relevant security problems, such as detecting network attacks or malware, proper labels are typically not available, resulting in a chicken-and-egg problem. As a remedy, researchers often revert to heuristics, such as using external sources that do not provide a reliable ground truth. For example, services like *VirusTotal* [115] are commonly used for acquiring label information for malware. Additionally, changes in adversary behavior may alter the ratio between different classes over time [194, 4, 327], introducing another type of bias known as *label shift* [176]. A system that

cannot adapt to these changes will experience performance decay once deployed.

Recommendations. Generally, labels should be verified whenever possible, for instance with sanity checks by inspecting a sample of labels [*e.g.*, 273]. If *noisy labels* cannot be ruled out, the impact of noisy labels on the resulting model can be reduced by (i) using robust models or loss functions by design, (ii) actively incorporating noisy labels by modeling them in the learning process, or (iii) cleaning the training data from noisy instances that increase the complexity of the model [see 104]. However, it should be stressed that instances with uncertain labels must not be removed from the test data. This represents a variation of sampling bias (P1) and data snooping (P3). Additionally, as labels are often subject to change over time in security settings it is necessary to check for *label shift* [176] and take precautions, such as delaying labeling until a stable ground-truth is available [see 327].

...but label noise can be modeled or otherwise accounted for by the system.

P3) Data Snooping. *A learning model is trained with data that is typically not available in practice. Data snooping can occur in many ways, some of which are very subtle and hard to identify* [1].

Description. It is common practice to split collected data into separate training and test sets prior to generating a learning model. Although splitting the data seems straightforward, there are many subtle ways in which test data (or other background information that is not usually available) can affect the training process, leading to data snooping. We broadly distinguish between three types of data snooping: *test*, *temporal*, and *selective snooping*.

Data snooping occurs when information from test time is used during design and training...

Test snooping occurs when the test set is used for experiments before the final evaluation. This includes preparatory work to identify useful features, parameters, and learning algorithms. Temporal snooping occurs if time dependencies within the data are ignored such as when C1 is violated. This is a common pitfall, as the underlying distributions in many security-related problems are under continuous change [e.g. 182, 223]. Finally, selective snooping describes the cleansing of data based on information not available in practice. An example is the removal of outliers based on statistics of the complete dataset (i.e., training and test) that are usually not available at training time.

Security implications. In security, data distributions are often non-stationary and continuously changing due to new attacks or technologies. Because of this, snooping on data from the future or from external data sources is a prevalent pitfall that leads to over-optimistic results. For instance, several researchers have identified temporal snooping in learning-based malware detection systems [*e.g.*, 6, 14, 223]. In all these cases, the capabilities of the methods are overestimated due to mixing samples from past and

present. Similarly, there are incidents of test and selective snooping in security research that lead to unintentionally biased results (see Section 5.10.3).

...satisfying C1, ensuring a final test set is kept completely separate...

Recommendations. While it seems obvious that training, validation, and test data should be strictly separated in all experiments, this separation is often unintentionally violated during the preprocessing stage of machine learning workflows. For example, we observe that it is a common mistake to compute tf-idf weights or neural embeddings over the entire dataset (see Section 5.10.3). To avoid this problem, test data should be split early during data collection and stored separately until the final evaluation. Furthermore, temporal dependencies within the data should be considered when creating the dataset splits [182, 6, 223]. However, there also exist more subtle variants of data snooping. For instance, as the characteristics of publicly available datasets are increasingly exposed, methods developed using this data implicitly leverage knowledge from the test data [see 190, 1].

Consequently, well-known datasets should be mainly used for comparison with past research and complemented with recent data from the application setting. In any case, it should always be discussed by researchers if the test data set may have influenced the results due to the evaluation procedure (*e.g.*, if a method has been developed and evaluated using only well-known datasets).

...and using fresh data can all help prevent snooping.

P4) False Causality. *Artifacts unrelated to the security problem create shortcut patterns for separating classes. Consequently, the learning model adapts to these artifacts instead of solving the actual task.*

Description. Data can contain artifacts that may loosely correlate with the task to solve but are not actually related to it. Consider the example of a network intrusion detection system, where a large fraction of the attacks in the dataset originate from a certain network region. The model may learn to detect a specific IP range instead of generic attack patterns. Similarly, a detection system might pick up artifacts from synthetic attacks that are unrelated to malicious activity, as in the classic case of the "why six?" issue [284].

False causality can lead to inflated performance which is actually due to data artifacts...

Security implications. Complex learning models with difficult to interpret feature spaces are often at the core of security tasks. Difficulties in explaining models and results leads to false causality, which often remains an unidentified issue.

...but these can often be identified and removed by using explainability techniques.

Recommendations. Learned artifacts are likely to hinder the successful application of the learning model in practice. Hence, explainable learning techniques should be used as a mandatory check [see 122, 162, 305]. These can reveal if the classification relies on spurious features.

P5) Biased Parameter Selection. *The final parameters of a learning-based method are not entirely fixed at training time. Instead, they indirectly depend on the test set.*

Description. Throughout the learning procedure, it is common practice to generate different models by varying hyperparameters. The best-performing model is picked and its performance on the test set is presented. While this setup may appear sound, it can still suffer from bias.

For example, misleading results may be produced by using uncalibrated metrics or by investigating the influence of hyperparameters on the test data.

Parameter selection can be biased if calibration is performed using test data...

Security implications. A security system whose parameters have not been fully calibrated at training time can perform very differently in a realistic setting. While the detection threshold for a network intrusion detection system may be chosen using a ROC curve obtained on the test set, it can be hard to select the same operational point in practice due the diversity of real-world traffic [267]. This may lead to decreased performance of the system in comparison to the original experimental setting. Note that this pitfall is related to data snooping (P3), but should be considered explicitly as it can easily lead to inflated results.

...but ensuring test data is kept isolated from tuning, calibration, and repeated experimentation can prevent this.

Recommendations. A recurring issue in security research is that the calibration is unintentionally performed on test data, for example, when the operating point of a system is chosen after all experiments have been completed. As this pitfall can be regarded as a special case of data snooping, the same countermeasures apply. To avoid this pitfall, it is of utmost importance to consequently use a separate *validation set* for all model selection and parameter tuning.

5.10.2 *Performance Evaluation*

The next stage in a typical machine-learning workflow is the evaluation of the system's performance. In the following, we show how different pitfalls can lead to unfair comparisons and biased results in the evaluation of such systems.

Inappropriate baselines are those that are too limited to justify the use of the new proposal...

P6) Inappropriate Baseline. *The evaluation is conducted without, or with limited, baseline methods. As a result, it is impossible to demonstrate improvements against the state of the art and other security mechanisms.*

Description. To show to what extent a novel method improves the state of the art, it is vital to compare it with previously proposed methods. When choosing baselines, it is important to remember that, despite great leaps forward in other fields, there exists no universal algorithm in machine learning that dominates all other

approaches in general [312]. Consequently, providing only results for the proposed approach or comparing it only with closely related methods, does not give enough context to assess its impact.

...including a comparison against both state-of-the-art methods and simple, non-ML methods ensures the bar is set fairly.

Security implications. An overly complex learning method does not only increase the chances of overfitting, but it also increases the runtime overhead, the attack surface, and the time and costs for deployment. To show that machine learning techniques provide significant improvements compared to traditional methods, it is essential to compare these systems side by side.

Recommendations. Instead of focusing solely on complex models for comparison, simple models should also be considered throughout the evaluation. These methods are easier to explain, less computationally demanding, and have proven to be effective and scalable in practice. Using well-understood, simple models as a baseline can expose unnecessarily complex learning models and automated machine learning (*AutoML*) frameworks [e.g., 96, 138] are a useful method for finding proper baselines. These frameworks enable researchers to automatically retrieve machine learning models that have been trained using state-of-the-art techniques for hyperparameter tuning and model selection. While these automated methods can certainly not replace experienced data analysts, they can be used to set the bar the proposed approach should aim for. Finally, it is critical to check whether non-learning approaches are also suitable for the application scenario. For example, for intrusion and malware detection, there exist a wide range of methods using other detection strategies [e.g., 219, 239, 87].

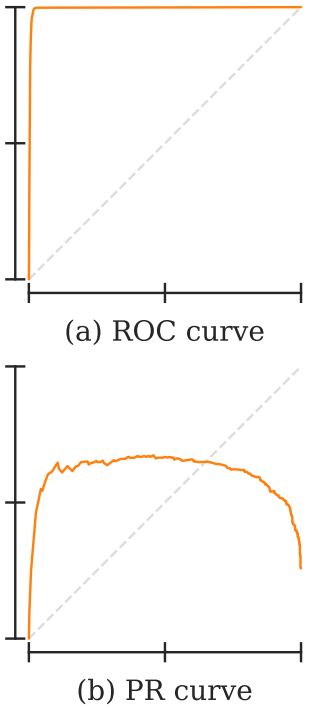


Figure 5.14: ROC and PR curves applied to results on an artificial dataset with an imbalanced class ratio. While the classifier's decision scores are the same in both cases, only the PR curve conveys the true performance.

P7) Inappropriate Performance Measures. *The chosen performance measures do not account for the constraints of the application scenario, such as imbalanced data or the need to keep a low false-positive rate.*

Description. A wide range of performance measures are available and not all of them are suitable in the context of security. For example, when evaluating a detection system, it is insufficient to report just a single performance value, such as the accuracy, because true-positive and false-positive decisions are not observable. However, even more advanced measures, such as ROC curves, may obscure experimental results. Figure 5.14 shows an ROC curve and a precision-recall curve on an imbalanced dataset (class ratio 1:100). Given the ROC curve alone, the performance appears excellent, yet the low precision reveals the true performance of the classifier.

Furthermore, various security-related problems deal with more than two classes, requiring *multi-class metrics*. This setting can introduce further subtle pitfalls. Common strategies, such as *macro-averaging* or *micro-averaging* are known to overestimate and underestimate small classes [101].

Using inappropriate performance metrics, such as AUC on imbalanced data, can give misleading results.

Security implications. Inappropriate metrics are a long-standing problem in security research, particularly in detection tasks. While true and false positives provide a more detailed picture of a system's performance, they can also disguise the actual precision when the prevalence of attacks is low.

Recommendations. As the choice of metrics is highly application-specific, we refrain from providing general guidelines. Instead, we recommend ensuring the chosen measures would help a practitioner assess the performance of the security system during a deployment (see P9).

P8) Base Rate Fallacy. *A large class imbalance is ignored when interpreting the performance measures leading to an overestimation of performance.*

Description. Class imbalance can easily lead to a misinterpretation of performance if the base rate of the negative class is not considered. If this class is predominant, even a very low false-positive rate can result in surprisingly high numbers of false positives. Note the difference to the previous pitfall: while P7 refers to the inappropriate *description* of performance, the base-rate fallacy is about the misleading *interpretation* of results. This special case is easily overlooked in practice (see Section 5.10.3). Consider the example in Figure 5.14 where 99% true positives are possible at 1% false positives. Yet, if we consider thote class ratio of 1:100, this actually corresponds to 100 false positives for every 99 true positives.

Ignoring the realistic base rate of the positive class can lead to erroneous interpretations of results...

Security implications The base rate fallacy is relevant in a variety of security problems, such as intrusion detection and website fingerprinting [*e.g.,* 23, 140, 210]. In website fingerprinting, users can visit billions of web pages, but only a tiny fraction of these web pages are available for evaluation. As a result, it is challenging to provide realistic numbers on the privacy threat posed by attackers. Similarly, the probability of installing malware is usually much lower than is considered in experiments on malware detection [223].

Recommendations. Several problems in security revolve around detecting a rare event, namely attacks, so we advocate the use of *precision* and *recall* as well as related measures, such as precision-recall curves. In contrast to several other performance measures, these functions do account for class imbalance and the base rate fallacy, and thus resemble reliable performance indicators for detection tasks focusing on a minority class [266, 75]. However, note that precision and recall can also be misleading, for instance, if the prevalence of attacks is inflated due to spatial bias as we have discussed earlier in this chapter. In these cases, other metrics like *Matthews Correlation Coefficient (MCC)* are more suitable to assess the classifier's performance and reveal potential weaknesses [61].

...but explicitly discussing the number of false positives will ensure a reader is not misled.

In addition, ROC curves are a useful metric for directly comparing the performance of multiple approaches, but their expressiveness depends highly on the selection of proper baselines. An explicit discussion of how the false positive rate of a proposed method relates to the base rate of the negative class allows readers to get a sound understanding of the system's capabilities in the practical use case.

P9) Lab-Only Evaluation. *A learning-based system is solely evaluated in a laboratory setting, without discussing its practical limitations.*

Description. As in all empirical disciplines, it is common to perform experiments under certain assumptions to demonstrate a method's efficacy. While performing controlled experiments is a legitimate way to examine specific aspects of an approach, it should ultimately be evaluated in a realistic setting to transparently assess its capabilities and showcase the open challenges which will foster further research.

Security implications. Many learning-based systems in security are evaluated solely in laboratory settings, overstating their practical impact. A common example are detection methods evaluated only in a *closed-world setting* with limited diversity and no consideration of non-stationarity [139, 27]. For example, a large number of website fingerprinting attacks are evaluated only in closed-world settings spanning a limited time period [140]. Similarly, many learning-based malware detection systems have been insufficiently examined in realistic settings as shown earlier.

Lab-only experiments do not always effectively simulate the deployment scenario...

Recommendations. It is essential to move from a *laboratory setting* and approximate a *real-world setting* as accurately as possible. As discussed earlier in this chapter, temporal and spatial relations of the data should be considered to account for the typical dynamics encountered in the wild. Similarly, runtime and storage constraints should be analyzed under practical conditions [see 241, 297, 27]. Ideally, the proposed system should be deployed to uncover problems that are not observable in a lab-only environment, such as the diversity and complexity of real-world network traffic [see 267].

...and practical limitations may not become apparent until moving to a real-world setting.

P10) Inappropriate Threat Model. *The security of machine learning is not considered, exposing the system to a variety of attacks, such as poisoning and evasion attacks.*

Description Learning-based security systems operate in an hostile environment. Prior work in adversarial learning has revealed a considerable attack surface introduced by machine learning itself at all stages of the workflow [see 34, 217]. First, *membership inference attacks* undermine models' privacy, allowing an adversary to leak information of training examples by exploiting overfitting in deep neural networks [258]. Next, *preprocessing attacks* target the feature

Due to the hostile environment, security detectors are only as useful as they are robust...

extraction step to inject arbitrary inputs to the system which affect all further steps in the pipeline [315]. *Poisoning and backdoor attacks* tamper with the data to modify a model's behavior [35, 121]. *Model stealing* allows for a model to be approximated, leaking intellectual property and accelerating further attacks [291]. Finally, *adversarial examples* are inputs that allow an adversary to control the final prediction [37, 50].

Security implications. Neglecting to include adversarial influence in the threat model and evaluation is fatal as a system deployed in an adversarial environment which is not robust to adversaries will not be able to provide trustworthy, meaningful results. Additionally, failing to consider machine-learning related attacks will expose the system to an additional attack surface—aside from traditional security issues. For instance, an attacker can more easily evade a model that relies on only a few features than a properly regularized model that has been designed with security considerations in mind [77]. Furthermore, the *semantic gap* describes the discrepancy between extracted features and the corresponding object [303]. For example, an adversary can create adversarial examples of PDFs by injecting content into areas that a regular PDF reader ignores but that a PDF malware detector examines. It therefore becomes easy to manipulate the PDF feature vector while ensuring the inconspicuousness of the corresponding object.

Recommendations. In most fields of security where learning-based systems are used, we operate in an *adversarial environment*. Hence, threat models should be defined precisely and systems evaluated with respect to them. In most cases, it is necessary to assume an *adaptive adversary* that specifically targets the proposed systems and will search for and exploit weaknesses for evasion or manipulation. Similarly, it is helpful to consider the different stages of the machine learning workflow and investigate possible vulnerabilities [see 34, 217, 51, 76]. White-box attacks should be employed to consider a worst-case scenario, following Kerckhoff's principle [147] and security best practices. Ultimately, a security system is of little practical utility, if it can be easily circumvented and thus an evaluation of adversarial aspects is a mandatory component in security research.

...so they must be evaluated assuming an adaptive attacker.

5.10.3 *Pitfalls Prevalence*

Similar to our study estimating the prevalence of spatio-temporal bias in Section 5.8, we now survey a set of representative research works from the community to assess the prevalence of these pitfalls. To this end, we conduct a ten-year retrospective study, with an emphasis on the past 6 years, on 30 representative papers published at the top-4 conferences for security-related research in our

community: ACM CCS, IEEE S&P, USENIX Security, and NDSS.

All papers apply machine learning to different security tasks, encompassing a broad variety of topics. Note that research focusing on the security of machine learning or that does not apply machine learning at all is considered out of scope.

In particular, our sample of top-tier papers includes the following topics: malware detection [19, 187, 223, 70, 314, 271]; network intrusion detection [83, 256, 196, 259]; vulnerability discovery [80, 97, 98, 172]; website fingerprinting attacks [260, 238, 84, 210]; social network abuse [206, 42, 269]; binary code analysis [257, 63, 26]; code attribution [136, 2]; steganography [29]; online scams [149]; game bots [166]; and ad blocking [135]. Figure 5.15 shows a breakdown of the papers by year of publication.

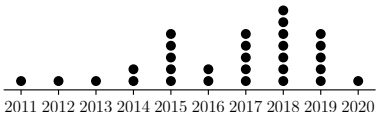


Figure 5.15: Distribution of papers per year for the 30 papers in our analysis.

Review process Each paper is assigned two independent reviewers who assess the article and identify instances of the described pitfalls. The pool of reviewers consists of six researchers who have all previously published work on the topic of machine learning and security in at least one of the considered security conferences. Reviewers do *not* consider any material presented outside the papers under analysis (other than their associated artifacts such as datasets or source code), and do *not* contact the authors for more information. Once both reviewers have completed their assignments, they discuss the paper in the presence of a third reviewer that may resolve any disputes. In case of uncertainty, the authors are given the benefit of the doubt (*e.g.*, in case of a dispute between *partly present* and *present*, we assign *partly present*).

To assess the prevalence of pitfalls in our community we review 30 top papers published since 2011...

Throughout the process, all reviewers meet regularly in order to discuss their findings and ensure consistency between the pitfalls' criteria. Moreover, these meetings have been used to refine the definitions and scope of pitfalls based on the reviewers' experience. Following any adaptation of the criteria, all completed reviews have been re-evaluated by the original reviewers—this occurred twice during our analysis. While cumbersome, this adaptive process of incorporating reviewer feedback ensures that the pitfalls are comprehensive in describing core issues across the state of the art.

We note that the inter-rater reliability of reviews prior to dispute resolution is $\alpha = 0.832$ using Krippendorff's alpha, where $\alpha > 0.800$ indicates confidently reliable ratings [155].

Assessment criteria For each paper, pitfalls are coarsely classified as either *present*, *not present*, *unclear from text*, or *does not apply*. A pitfall may be wholly present throughout the experiments without remediation *(present)*, or it may not *(not present)*. If the authors have corrected any bias or have narrowed down their claims to accommodate the pitfall, this is also counted as *not present*. Additionally, we introduce *partly present* as a category to account for experiments that do suffer from a pitfall, but where the impact has been partially addressed. If a pitfall is *present* or *partly present* but ac-

...using a scale from present to not present, moderating the score if the pitfall was discussed.

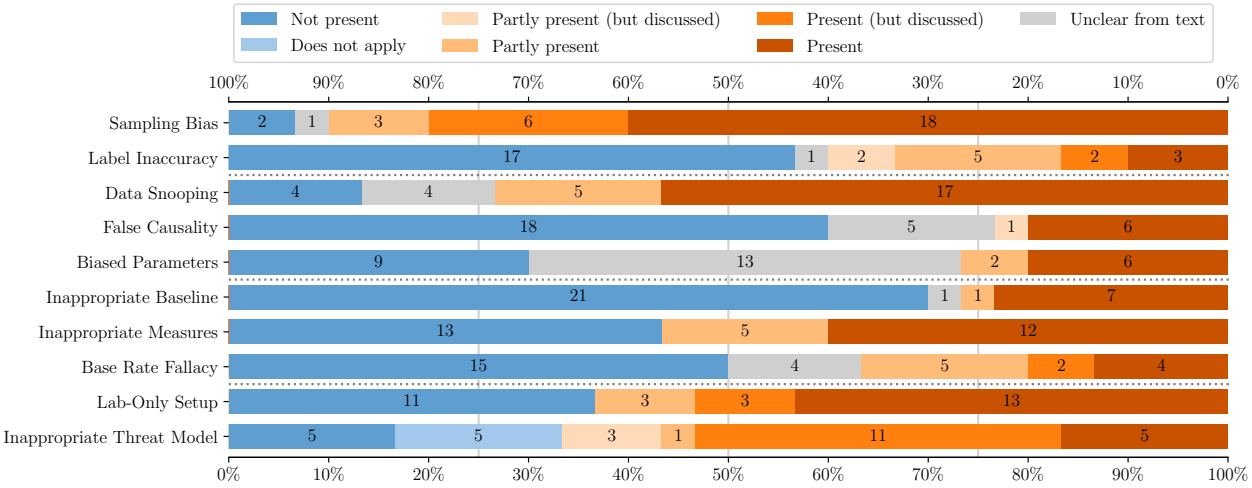


Figure 5.16: Stacked bar chart showing the pitfalls suffered by each of the 30 papers analyzed. The colors of each bar show the degree to which a pitfall was present, and the width shows the proportion of papers in that group. The number at the center of each bar shows the cardinality of each group.

knowledged in the text, we moderate the classification as *discussed*. If the reviewers are unable to rule out the presence of a pitfall due to missing information, we mark the publication as *unclear from text*. Finally, in the special case of P10, if the pitfall *does not apply* to a paper's setting, this is considered as a separate category.

Observations The aggregated results from the prevalence analysis are shown in Figure 5.16. A bar's color indicates the degree to which a pitfall is present, and its width shows the proportion of papers with that classification. The number of affected papers is noted at the center of the bars. The most prevalent pitfalls are sampling bias (P1) and data snooping (P3), which are at least partly present in over 73% of the papers. In more than 50% of the papers, we identify inappropriate threat models (P10), lab-only evaluations (P9), and inappropriate baselines (P6) as at least partly present. *Every* paper is affected by at least three pitfalls, underlining the pervasiveness of such issues in recent computer security research. In particular, we find that dataset collection is still very challenging: some of the most realistic and expansive open datasets we have developed as a community are still imperfect.

Moreover, the presence of some pitfalls is more likely to be *unclear from the text* than others. We observe this for biased parameter selection (P5) when no description of the hyperparameters or tuning procedure is given; for false causality (P4) when there is no attempt to explain a model's decisions; and for data snooping (P3) when the dataset splitting or normalization procedure is not explicitly described in the text. These issues also indicate that experimental settings are more difficult to reproduce due to a lack of information.

We find that all pitfalls are pervasive in security research affecting between 23% and 90% of papers.

Takeaways We find that all of the pitfalls are pervasive in security research, affecting between 23% and 90% of the selected papers.

Each paper suffers from at least three of the pitfalls which, compounded by the fact that only 20% of instances are accompanied by a discussion in the text, indicates a clear lack of awareness in our community.

Remark. Although our findings point to a serious problem in research, we would like to remark that *all* of the papers analyzed provide excellent contributions and valuable insights. Our objective here is not to blame researchers for stepping into pitfalls but to raise awareness and increase the experimental quality of research on machine learning in security.

5.11 Related Work

As described in the previous section, a common experimental bias in security is the *base rate fallacy* [23], which states that in highly-imbalanced datasets (*e.g.,* network intrusion detection, where most traffic is benign), TPR and FPR are misleading performance metrics, because even FPR $= 1\%$ may correspond to *millions* of FPs and only *thousands* of TPs. In contrast, our work identifies experimental settings that are misleading *regardless* of the adopted metrics, and that remain incorrect even if the right metrics are used (Section 5.6). Sommer and Paxson [267] discuss challenges and guidelines in ML-based intrusion detection; Rossow et al. [241] discuss best practices for conducting malware experiments; van der Kouwe et al. [296] identify 22 common errors in system security evaluations. While helpful, these works [296, 241, 267] do not identify temporal and spatial bias, do not *quantify* the impact of errors on classifiers performance, and their guidelines would not prevent all sources of temporal and spatial bias we identify. To be precise, Rossow et al. [241] evaluate the percentage of objects—in previously adopted datasets—that are "incorrect" (*e.g.,* goodware labeled as malware, malfunctioning malware), but without evaluating impact on classifier performance. Zhou et al. [326] have shown that Hardware Performance Counters (HPCs) are ineffective for malware classification; while interesting and in line with the spirit of our work, their focus is narrow, and they rely on 10-fold CV in their evaluation.

Allix et al. [6] broke new ground by evaluating malware classifiers in relation to time and showing how future knowledge can inflate performance, but do not propose any solution for comparable evaluations and only identify C1. As a separate issue, Allix et al. [7] investigated the difference between in-the-lab and in-the-wild scenarios and found that the greater presence of goodware leads to lower performance. We systematically analyze and explain these issues and help address them by formalizing a set of constraints (jointly considering the impact of temporal and spatial bias), introducing AUT as a unified performance metric for fair

time-aware comparisons of different solutions, and offering a tuning algorithm to leverage the effects of training data distribution. Miller et al. [194] identified *temporal sample consistency* (equivalent to our constraint C1), but not C2 or C3—which are fundamental (Section 5.6); moreover, they considered the test period to be a uniform time slot, whereas we take time decay into account. Roy et al. [242] questioned the use of recent or older malware as training objects and the performance degradation in testing real-world object ratios; however, most experiments were designed without considering time, reducing the reliability of their conclusions. While past work highlighted some sources of experimental bias [194, 242, 6, 7], it also gave little consideration to classifiers' aims: different scenarios may have different goals (not necessarily maximizing F_1), hence in our work we show the effects of different training settings on performance goals and propose an algorithm to properly tune a classifier accordingly (Section 5.5.3).

Other works from the ML literature investigate imbalanced datasets and highlighted how training and testing ratios can influence the results of an algorithm [309, 126, 57]. However, not coming from the security domain, these studies [309, 126, 57] focus only on some aspects of spatial bias and do *not* consider temporal bias. Indeed, concept drift is less problematic in some applications (*e.g.,* image and text classification) than in Android malware [139]. Fawcett [93] focuses on challenges in spam detection, one of which resembles spatial bias; no solution is provided, whereas we introduce C3 to this end and demonstrate how its violation inflates performance (Section 5.6). Torralba and Efros [289] discuss the problem of *dataset bias* in computer vision, distinct from our security setting where there are fewer benchmarks; moreover in images the negative class (*e.g.,* "not cat") can grow arbitrarily, which is less likely in the malware context. Moreno-Torres et al. [199] systematize different *drifts*, and mention *sample-selection bias*; while this resembles spatial bias, they do not propose any solution/experiments for its impact on ML performance. Other related work underlines the importance of choosing appropriate performance metrics to avoid an incorrect interpretation of the results (*e.g.,* ROC curves are misleading in an imbalanced dataset [124, 75]). In this chapter, we take imbalance into account, and we propose actionable constraints and metrics with tool support to evaluate performance decay of classifiers over time.

Overall, several studies of bias exist and have motivated our research, but none address the entire problem in the context of evolving data (where the i.i.d. assumption does not hold anymore). Constraint C1, introduced by Miller et al. [194], is by itself insufficient to eliminate bias. This is evident from the original evaluation in MaMaDroid [187], which enforces only C1. The evaluation in Section 5.6 clarifies why our novel constraints C2 and C3 are fundamental, and shows how our AUT metric can effectively reveal the true performance of algorithms, providing counterintuitive results.

5.12 Summary

We identify novel sources of temporal and spatial bias in the Android domain and propose novel constraints, metrics, and tuning to address such issues. We build and release TESSERACT as an open-source tool that integrates our methods. We show how TESSERACT can reveal the real performance of malware classifiers that remain hidden in wrong experimental settings in a non-stationary context. TESSERACT is fundamental for the correct evaluation and comparison of different solutions, in particular when considering mitigation strategies for time decay and quantifying the impact of concept drift. Beyond these sources of bias we additionally describe common *pitfalls* in the evaluation of machine learning-based methods applied to the security domain and measure their prevalence in recent research. Addressing these sources of bias is essential to ensure the stable and consistent evaluation of methods for mitigating concept drift and adversarial examples.

6 *Identifying and Rejecting Drifting Examples*

It is clear that concept drift is a major obstacle towards the successful deployment of machine-learning based security detectors. Previous chapters have illustrated the extent to which drifting and adversarial examples impact detection performance and how to design consistent and realistic experiments to fairly evaluate potential defenses against them.

In this chapter we focus on one promising strategy in particular: classification with a reject option. Having a reject option allows a classifier to discard low quality predictions that are likely to be incorrect—such as those made for drifting examples. Additionally, identifying and tracking drifting examples allows us to monitor the changing character of the data distribution over time and better understand the factors contributing to drift.

Chapter 5 already touched on a naïve application of classification with rejection, but here we examine a more sophisticated method: Transcendent, a rejection framework that builds on Transcend [139], and relies on *conformal prediction* and *conformal evaluation* theory to identify and reject drifting examples.

6.1 Key Insights

For reference, this chapter provides the following contributions:

- We investigate the theory underpinning the motivation and intuition of conformal evaluation to provide a missing link between conformal evaluation and conformal prediction theory that explains its effectiveness and supports the empirical evaluations presented in both this work and the original (Section 6.4).
- Building on this insight, we propose two novel conformal evaluators: *inductive conformal evaluator* (ICE) (Section 6.5.2) and *cross-conformal evaluator* (CCE) (Section 6.5.3), both of which are firmly grounded in conformal prediction theory and able to effectively identify and reject drifting examples while being significantly less computationally demanding than the original. We formal-

ize the calibration procedure as an optimization problem and propose an improved threshold search strategy (Section 6.6).

- We evaluate our proposals on a dataset spanning 5 years (2014–2019) containing ~10% malware that eliminates sources of bias present in past evaluations (Section 6.7). We compare different operational settings, including the effects of including algorithm *confidence* (Section 6.7.3) and of using different search strategies (Section 6.7.4) during thresholding. Our methods outperform existing state-of-the-art approaches (Section 6.7.5), and generalize well across different malware domains and underlying classifiers (Section 6.7.6). To aid practitioners in adopting rejection strategies, we discuss how to integrate Transcendent into a typical security detection pipeline (Section 6.8).

The content of this chapter has been previously presented in:

- Barbero F.*, Pendlebury F.*, Pierazzi F., Cavallaro L. Transcending TRANSCEND: Revisiting Malware Classification in the Presence of Concept Drift. To appear in *In Proc. of the IEEE Symposium of Security and Privacy (S&P)*. 2022.

6.2 Overview

While machine learning (ML) algorithms have displayed superhuman performance across a wide range of classification tasks such as computer vision [156] and natural language processing [86], a great deal of this success is conditional on one central assumption: that the training and test data are drawn identically and independently from the same underlying distribution (i.i.d.) [40].

Machine learning relies on the i.i.d. assumption which is violated in security settings...

To reiterate the central theme of this thesis: in a security setting the i.i.d. assumption often does not hold. In particular, malware classifiers are deployed in dynamic, hostile environments. New paradigms of malware evolve to pursue profits, new variants arise as novel exploits are discovered, and adversaries switch behavior suddenly and dramatically when faced with strengthened defenses. This causes the incoming test distribution to diverge from the original training distribution, a phenomenon known as *concept drift* [199]. Over time, the performance of the classifier begins to degrade as the model fails to classify the new objects correctly.

There appear to be two broad approaches to tackling concept drift. The first is to design systems which are intrinsically more resilient by developing more robust feature spaces. For example, results in the previous chapter suggest that neural networks may be more resilient to concept drift as the latent feature space better generalizes to new variants. However, robust feature space design is an open research question and it is not clear if there exists a malware representation such that no concept drift will not occur.

...but it is possible to adapt to the new distribution if drift can be accurately identified...

A second solution is to *adapt* to the drift, for example by updating the model using incremental retraining or online learn-

ing [316, 204], or rejecting drifting points. However, in order to be effective, decisions about when and how to take action on aging classifiers must be taken quickly and decisively. To do so, accurate detection and quantification of drift is vital.

...which was the focus of Transcend [139], a framework using conformal evaluation...

This problem is precisely the focus of Transcend [139], a statistical framework that builds on conformal prediction theory [301] to detect aging malware detectors during deployment—before their accuracy deteriorates to unacceptable levels. Transcend [139] proposes a *conformal evaluator* that utilizes the notion of *nonconformity* to identify and reject new examples that differ from the training distribution and are likely to be misclassified; the corresponding apps can then be quarantined for further analysis and labeling. While effective, the original proposal suffers from experimental bias, is extremely resource intensive and impractical, lacks experiments to support generalization claims, fails to provide guidance on how to integrate it into a detection pipeline and, perhaps more importantly, lacks a theoretical analysis to explain its effectiveness.

...and is revisited in this work as Transcendent, an improvement in terms of effectiveness, cost, and generalizability.

We revisit conformal evaluator and Transcend to root its internal workings in sound theory and determine its operational settings. We additionally propose Transcendent, a framework that surpasses the performance of the original in terms of drift detection and computational overhead, making it a sound and practical solution.

6.3 Concept Drift and Rejection

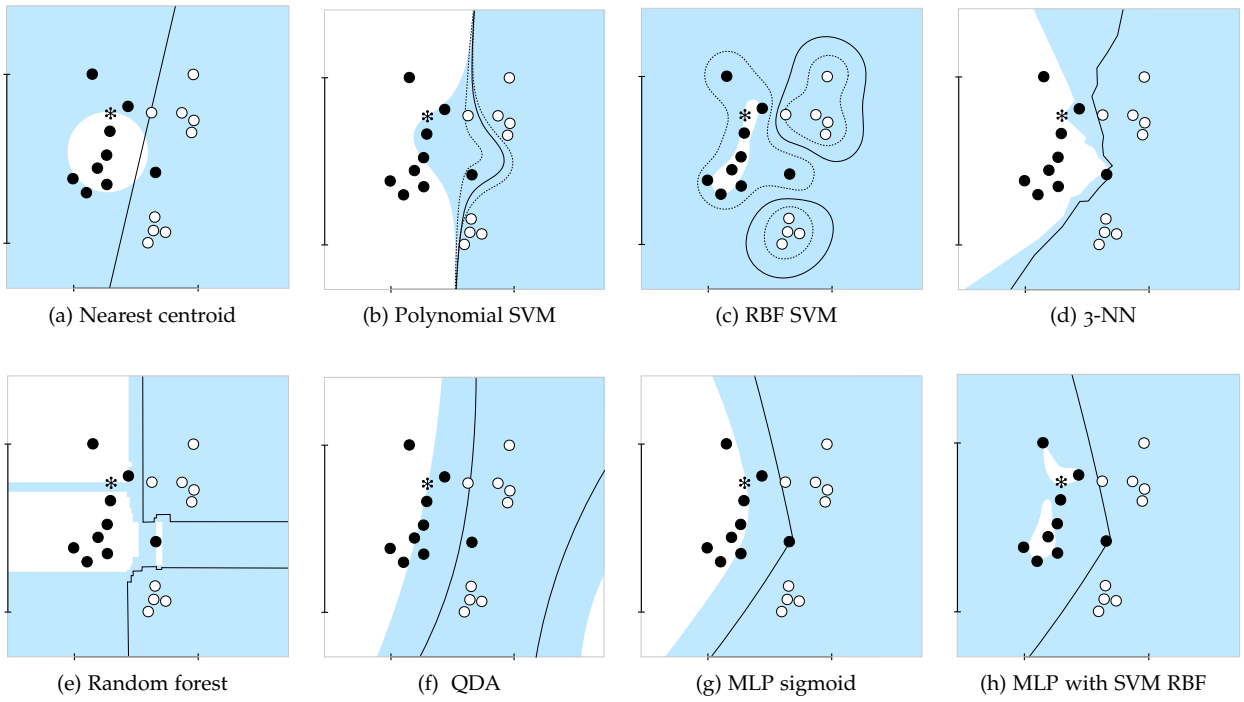
We focus on classification for security tasks (Section 6.3.1) which are affected by concept drift (Section 6.3.2). In particular, we are interested in improving the state-of-the-art approaches for classification with rejection (Section 6.3.3).

6.3.1 *Machine Learning and Security Detection*

Machine learning is a set of statistical methods for enabling systems to perform data-driven tasks without being explicitly programmed for them. In the malware domain, typical tasks include binary classification (detecting malicious examples [19, 316]) and multiclass classification (predicting the malware family [275, 74, 276]) but can also extend to more complex tasks such as predicting how many AV engines would detect an example [145], inferring Android malware app permissions based on their icons [314], or generating Windows malware using reinforcement learning [12].

As before we focus on drift in classification tasks.

As in previous chapters, here we focus on classification tasks where a classifier g aims to learn a function mapping $\mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} \subseteq \mathbb{R}^n$ is a feature space of vectors capturing interesting properties of the apps and $\mathcal{Y}$ is a label space containing binary labels for the detection task or the names of malware families for the multiclass classification task.



(a) Nearest centroid (b) Polynomial SVM (c) RBF SVM (d) 3-NN

(e) Random forest (f) QDA (g) MLP sigmoid (h) MLP with SVM RBF

Figure 6.1: Possible NCMs for different classifiers. The solid line depicts the decision boundary between classes ● and ○, the dotted lines show SVM margins. Shaded areas capture points which are *more nonconform* (*i.e.*, 'less similar') than the new test point (∗), with respect to class ●.

6.3.2 *Concept Drift*

As we have described in Chapter 3, concept drift is a common phenomenon in classification tasks when the joint distribution of inputs and outputs differs between training and test time [230]. This drift induces a performance decay over time as the model loses the ability to distinguish between members of each class—this performance decay was the focus of Chapter 5.

Sources of drift in malware classification can be fairly benign, such as changes in market trends or new developer APIs [325]. However, the main driving force of drift is the development of new malware techniques to evade detection [4, 12, 226, 318], increase infection rates [283], and generate greater profits [148].

6.3.3 *Rejection*

In lieu of robust feature spaces, drift should be tracked and mitigated...

There are multiple routes to dealing with concept drift. The most effective would be to design a feature space $\mathcal{X}$ such that it is entirely robust to concept drift, essentially distilling all possible malware behaviour down to a 'Platonic ideal' [229] that captures maliciousness no matter what form it takes. While recent proposals for augmenting feature spaces with robust features are promising [*e.g.*, 288, 325], the diversity of malware makes it extremely difficult to design such a feature space. Additionally, some behaviour is only considered malicious due to its context, for example, requesting access to the device contacts might be considered suspicious for a torch app but not for a social messaging app [320].

An orthogonal approach is to identify, track, and mitigate the

drift as it occurs. One promising method is classification with rejection [30], in which low confidence predictions, caused by drifting examples, are *rejected*. Drifting apps can then be quarantined and dealt with separately, either warranting manual inspection or remediation through other means.

Transcend [139] is a state-of-the-art framework for performing classification with rejection in security tasks. It uses a *conformal evaluator* to generate a quality measure to assess whether a new test example is drifting with respect to the training data. If the prediction of an underlying classifier appears to be affected by the drift, the prediction is rejected. The original proposal presented two case studies: Android malware detection—a binary classification task, and Windows malware family classification—a multiclass classification task. The experiments showed that the framework is consistently able to identify drifting examples, providing a significant improvement over thresholding on the classifiers' output probabilities. However, the lack of a theoretical treatment and the computational complexity of the framework limited its understanding and use in real-world deployments.

...which Transcend [139] achieves through classification with rejection.

6.4 Towards Sound Conformal Evaluation

The statistical engine that drives Transcend's rejection mechanism is the *conformal evaluator*, a tool for measuring the quality of predictions output by an underlying classifier. Conformal evaluator design is grounded in the theory of *conformal prediction* [301], a method for providing predictions that are correct with some guaranteed confidence. In this section we investigate the relationship between the two to provide novel insights and intuition into why conformal evaluation is effective for classification with rejection.

Transcend [139] is driven by a conformal evaluator...

6.4.1 *Conformal Evaluation vs. Prediction*

Here we give an overview of conformal prediction and how it motivates the use of conformal evaluation; for a more formal treatment of conformal prediction we refer to Vovk et al. [301]. Conformal prediction allows for predictions to be made with precise levels of confidence by using past experience to account for uncertainty. Given a classifier g, a new example $z = (x, y)$, and a significance level ε, a conformal predictor produces a *prediction region*: a set of labels in the label space $\mathcal{Y}$ that is guaranteed to contain the correct label y with probability no more than $1 - \varepsilon$. To calculate this label set, the conformal predictor relies on a *nonconformity measure* (NCM) derived from g and uses it to generate scores representing how *dissimilar* each example is from previous examples of each class. To quantify this relative dissimilarity, *p-values* are calculated by comparing the nonconformity scores between ex-

...which adapts conformal prediction theory to drifting settings...

amples (Section 6.4.2). As well as these p-values, two important metrics are derived from the prediction region, confidence and credibility (Section 6.4.3), which can be used to judge the effectiveness of the conformal prediction framework. Conformal predictors are able to make strong guarantees on the correctness of each prediction so long as two assumptions about new test examples hold: the *exchangeability* assumption, that the sequence of examples is exchangeable, a generalization of the i.i.d. property; and the *closed-world* assumption, that new examples belong to one of the classes observed during training.

Rather than making predictions, conformal evaluators [139] borrow the same statistical tools (*i.e.,* nonconformity measures and p-values) but use them to *evaluate* the quality of the prediction made by the underlying classifier g. By detecting instances which appear to violate the aforementioned assumptions they can, with high confidence, *reject* new drifting examples which would otherwise be at risk of being misclassified.

...in order to assess the reliability of (and possibly reject) a prediction.

6.4.2 Nonconformity Measures and P-values

In order to reject a new example that cannot be reliably classified, conformal evaluators rely on a notion of *nonconformity* to quantify how dissimilar the new example is to a history of past examples. In general, a *nonconformity measure* (NCM) [253] is a real-valued function that outputs a score describing how different an example z is from a bag of previous examples $B = \{z_1, z_2, ..., z_n\}$:

$$\alpha_z = A(B, z). \tag{6.1}$$

The greater the value of α_z, the less similar z is to the elements of the bag B. An NCM is typically formed of two components: a metric $d(z, z')$ to measure the distance between two points, and a *point predictor* $\hat{z}(B)$ to represent B:

$$A(B, z) := d(\hat{z}(B), z). \tag{6.2}$$

A nonconformity measure (NCM) outputs a score of how dissimilar a new object is to a given class...

Illustrating this, Figure 6.1(a) shows an NCM for a nearest centroid classifier in which the Euclidean distance is used for $d(z, z')$, and the nearest class centroid is used for $\hat{z}(B)$.

For a new example z^*, the conformal evaluator must decide whether or not to approve the null hypothesis asserting that z^* *does not belong* in the prediction region formed by elements of B. To perform such a hypothesis test, *p-values* are calculated using the NCM values for each point. First the nonconformity score of z^* must be computed (Equation 6.3) along with nonconformity scores of elements in B (Equation 6.4), then the the p-value p_{z^*} for z^* is given as the proportion of points with greater or equal

nonconformity scores (Equation 6.5):

$$\alpha_{z^*} = A(B, z^*) \tag{6.3}$$

$$S = \{A(B \setminus \{z\}, z) : z \in B\} \tag{6.4}$$

$$p_{z^*} = \frac{|\alpha \in S : \alpha >= \alpha_{z^*}|}{|S|} \tag{6.5}$$

In the classification context, we can calculate p-values in a *label conditional* manner, such that B contains only previous examples of class $\hat{y} \in Y$ where $\hat{y} = g(z^*)$ is the predicted class of the new example. If p_{z^*} falls above a given significance level the null hypothesis is disproved and $\hat{y}$ is accepted as a valid prediction. Transcend [139] computes *per-class thresholds* to use as significance levels (Section 6.6).

...and used to compute p-values which are compared to a per-class threshold indicating whether the corresponding prediction should be rejected.

As p-values are calculated by considering nonconformity scores relative to one another, NCMs can be transformed monotonically without any impact on the resulting p-values. Thus, when designing an NCM in the form given by Equation 6.2, the distance metric $d(z, z')$ is significantly less important than the point predictor $\hat{z}(B)$. It is important to note that conformal evaluator algorithms are agnostic to the underlying NCM chosen, but the quality of the NCM—and particularly of $\hat{z}(B)$, will impact the ability of conformal evaluators to discriminate between valid and invalid predictions [253].

An *alpha assessment* [139] can be used to empirically evaluate how appropriate an NCM is for a given dataset by plotting the distribution of p-values for each class, further split into whether the prediction was correct or incorrect. As incorrect predictions should be rejected, they are expected to fall below the threshold, while correct predictions are expected to fall above the threshold. Well-separated distributions of correct and incorrect predictions suggest a viable threshold exists to separate them at test time. Poorly separated prediction p-values indicate an inappropriate NCM. An example of an alpha assessment on a toy dataset is shown in Figure 6.4 (d).

A good NCM should allow p-values of correct and incorrect predictions to be easily separated...

Figure 6.1 illustrates possible NCMs for different algorithms on a toy binary classification task with existing class examples ●/○ and new test example ∗.

The different algorithms illustrated are nearest centroid, support-vector machines (SVMs), nearest neighbors (NN), random forest, quadratic discriminant analysis (QDA), and multilayer perceptron (MLP). The solid line delineates the decision boundary between classes ● and ○ while the dotted lines show SVM margins where applicable. The shaded region captures points which are *more nonconform* (*i.e.*, 'less similar') than the new test point, with respect to class ●. As NCMs, (a) uses the distance from the class centroid; (b) and (c) use the negated absolute distance from the hyperplane; (d) uses the proportion of nearest neighbors belonging to class ○; (e) uses the proportion of decision trees that predict ○; (f) uses the negated probability of belonging to class ●; (g) uses the negated probability output by the final sigmoid activation layer; (h) uses

...but can be designed for many different learning algorithms.

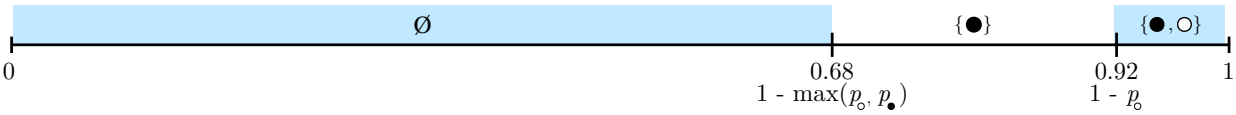


Figure 6.2: Nested intervals for which the output set contains labels ● and ○ for a test example with per-class p-values $p_{\bullet} = 0.32$ and $p_{\circ} = 0.08$. The shaded areas outline how credibility and confidence relate to the intersection of prediction regions for which the label set contains one element. The high probability of the empty set containing the correct label (*i.e.*, low credibility) indicates one of conformal prediction's assumptions may have been violated. Conformal evaluation uses this as a signal that the new example is drifting.

the outputs of the final hidden layer to train an SVM with RBF kernel and uses the negated absolute probabilities output by that SVM—while the decision boundary still depends on the MLP output alone).

Note that the shape of the nonconformal region need not reflect the shape of the regions for the predicted classes (*e.g.*, (a)) and that there may be multiple viable NCMs for the same underlying algorithm (*e.g.*, (g–h)).

6.4.3 *Successfully Identifying Drift*

Recall that conformal prediction produces a prediction region given a significance level ε. The possible prediction regions are nested such that the higher the confidence level, the more labels will be present. As a trivial example, a prediction region containing all possible labels may be produced for a significance level of $\varepsilon = 0$ (maximum likelihood) as it will contain the true label y with certainty. At the other extreme, an empty set can be produced at a significance level of $\varepsilon = 1$ (minimum likelihood), as this is an impossible result under the closed-world assumption of conformal prediction.

Of particular interest is the prediction region containing a single element which lies between these extremes. Related to this prediction region, a conformal predictor also outputs two metrics: *confidence* and *credibility* (Figure 6.2).

Confidence is the greatest $1 - \varepsilon$ for which the prediction region contains a single label which can be calculated as the complement to 1 of the second highest computed p-value. Confidence quantifies the likelihood that the new element belongs to the predicted class.

Conformal prediction produces two quality metrics: confidence and credibility...

Credibility is the greatest ε for which the prediction region is empty and corresponds to the largest computed p-value. Conformal predictors can be forced to output single predictions (rather than a label set induced by ε), in which case they will output the class with the highest credibility. Credibility quantifies how relevant the training set is to the prediction. A low credibility indicates that conformal prediction might not be a suitable framework to use with the given data because a low credibility means that the probability of the correct label being in the empty set is relatively high, which is an impossible result under the closed-world assumption of conformal prediction.

We propose that conformal evaluation's effectiveness stems from this relationship: that in conformal *evaluation*, this probability is being directly interpreted as the probability that the i.i.d. assumption has been violated. Thus, a low credibility means that there is

Conformal Evaluator	Complexity	Runtime in Section 6.7.2
TCE	$\mathcal{O}(n^2)$	est. 1.9 CPU yrs
Approx-TCE, $1/(1-p)$ folds	$\mathcal{O}(n/(1-p))$	46.1 CPU hrs
ICE	$\mathcal{O}(pn)$	11.5 CPU hrs
CCE, $1/(1-p)$ folds	$\mathcal{O}(pn/(1-p))$	36.6 CPU hrs

Table 6.1: Runtime complexities and empirical runtime for conformal evaluator calibration where n is the number of training examples and p is the proportion of examples included in the *proper training set* each split/fold.

a high probability that the corresponding example is *drifting* with respect to the previous history of training examples. Such an example is at risk of being misclassified due to limited knowledge of the classifier.

It should be noted that formally, conformal evaluation defines credibility and confidence slightly differently. In conformal evaluation, the credibility is the p-value corresponding to the predicted class and the confidence is the complement to 1 of the maximum p-value excluding the p-value corresponding to the predicted class (*i.e.*, the credibility p-value). This subtle difference is important to clarify the operational context of a conformal evaluator: whereas conformal predictors output the final classification decision, conformal evaluators output a statistical measure *separate* to the decision of the underlying classifier (hence the nomenclature: one predicts and the other evaluates). In practice, given reasonable NCMs, these definitions can be treated as equivalent.

...conformal evaluation interprets credibility as a signal that the i.i.d. assumption has been violated.

6.5 Towards Practical Conformal Evaluation

In assessing the quality of a prediction for a new test point, there is the question of which previously encountered points the new point should be compared to—that is, which elements are included in the bag B of Equation 6.3, and how. Typically, new test points are compared against a set of calibration points.

In Jordaney et al. [139], conformal evaluation was realized using a Transductive Conformal Evaluator (TCE). With a TCE, every training point is also used as a calibration point. To generate the p-value of a calibration point, it is first removed from the set of training points and the underlying classifier trained on the remaining points. Given the newly trained classifier, a predicted label is generated for the calibration point. Finally, using a given NCM, its p-value is computed with respect to the points whose ground truth label matches its predicted label. This procedure is repeated for every training point. Following this, Transcend's thresholding mechanism operates on the calculated p-values to determine per-class rejection thresholds (Section 6.6). At test time, the underlying classifier is retrained on the entire training set, and, similarly to the calibration points, the p-values are computed with respect to the p-values of the calibration sets.

Transcend [139] used a Transductive Conformal Evaluator (TCE) which was extremely computationally expensive...

While the Transductive Conformal Evaluator (TCE) used in the original proposal [139] appears to perform well, it does not scale to larger datasets as a newly trained classifier is required for ev-

ery training point. Consider the experiments in Section 6.7 where fitting a single instance of the underlying classifier takes 10 CPU minutes. In this case, we estimate a single run using vanilla TCE to take 1.9 CPU years.

We propose a number of novel conformal evaluators that overcome this limitation and present their advantages and disadvantages. A comparison of their runtime complexities and operational considerations are presented in Table 6.1 and Section 6.8, respectively. Formal algorithms for their calibration and test procedures are included in Section 6.5.4 while Figure 6.3 provides a graphical intuition to their different calibration splits.

Note that while our illustrative examples and evaluation are given for the binary detection task, Transcendent and conformal evaluation are agnostic to the total number of classes and this is captured in the formal definitions. If multiclass NCMs cannot be derived, per-class conformal evaluators may be arranged as a *one-vs-all* ensemble.

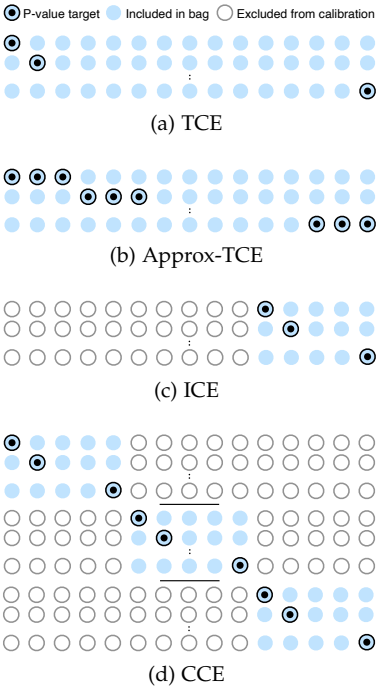


Figure 6.3: Illustration of the different calibration splits employed by each of the conformal evaluators showing the target of the p-value calculation, relative points included in the bag, and points excluded from the calibration.

6.5.1 *Approximate TCE (approx-TCE)*

Our first attempt at reducing the computational overhead induced by the Transductive Conformal Evaluator is the *approximate Transductive Conformal Evaluator* (approx-TCE). In the original TCE, p-values are generated for each calibration point by removing them from the training set, retraining the underlying classifier on the remaining points, and repeating until a p-value is computed for every training point.

In approx-TCE, calibration points are left out in batches, rather than individually. The training set is randomly partitioned into k folds of equal size. From the k folds, one is used as the target of the calibration and the remaining $k-1$ folds are used as the bag to which those points are compared to. This process repeats k times, until each fold has been used as the calibration set exactly once. Note that all of the k calibration sets are mutually exclusive; the corresponding batches of p-values are then concatenated in the same manner as in TCE.

...we propose a relaxed approximate variant (approx-TCE)...

The statistical soundness of the approx-TCE relies on the assumption that the decision boundary obtained from leaving out calibration points in batches approximates each of the decision boundaries that would have been obtained per calibration point in the batch if the point had been left out individually. If this assumption holds, the generated p-values will be the same as, or similar to, the p-values generated with a TCE. The approximation grows more accurate as k increases until k equals the cardinality of the training set at which point the approx-TCE and the TCE are equivalent. In this sense, the approx-TCE can be viewed as a generalization of the TCE.

This assumption is more likely to hold with algorithms with

lower variance (*e.g.*, linear models), but becomes more tenuous as the variance increases unless k increases also—sacrificing the saved computation to mitigate the statistical instability.

6.5.2 *Inductive Conformal Evaluator (ICE)*

...a new inductive conformal evaluator (ICE) which is more computationally efficient...

The second conformal evaluator we propose is the *Inductive Conformal Evaluator* (ICE) which, unlike the approx-TCE, is based on a corresponding approach from conformal prediction theory [301, 300, 211]. The ICE directly splits the training set into two non-empty partitions: the *proper training set* and the *calibration set*. The underlying algorithm is trained on the proper training set, and p-values are computed for each example in the calibration set. Unlike the TCE, p-values are not calculated for every training point, but only for examples in the calibration set, with the proper training set having no role in the calibration at all. The ICE aims to inductively learn a *general rule* on a single fold of the training set.

This induces significantly less computational overhead than the TCE and approx-TCE (see Table 6.1) and in practice is extremely fast, but also very *informationally inefficient*. Only a small proportion of the training data is used to calibrate the conformal evaluator, when ideally we would use all of it. Additionally, the performance of the evaluator depends heavily on the quality of the split and the calibration set's ability to generalize to the remainder of the dataset. This results in some uncertainty: an ICE may perform worse than a TCE due to a lack of information, or better due to a lucky split.

6.5.3 *Cross-Conformal Evaluator (CCE)*

...and a new cross-conformal evaluator (CCE) which is both computationally and informationally efficient, with extra flexibility.

The *Cross-Conformal Evaluator* (CCE) draws on inspiration from k-fold cross validation and aims to reduce both the computational and informational inefficiencies of the TCE and ICE. Like the ICE, the CCE has a counterpart rooted in conformal prediction theory [302].

The training set is partitioned into k folds of equal size. So that a p-value is obtained for every training example, each fold is treated as the calibration set in turn, with p-values calculated as with an ICE, using the union of the $k-1$ remaining folds as the proper training set to fit the underlying classifier.

Finally we concatenate the p-values in a way which preserves their statistical integrity when decision quality is evaluated. We set aside the k fit underlying models and corresponding calibration sets for test time. When a new point arrives, the prediction from each classifier is evaluated against the corresponding calibration set. The final result is the majority vote over the k folds, *i.e.*, the prediction of a particular class is accepted if the number of accepted classifications is greater than $\frac{k}{2}$, and rejected otherwise.

The CCE can be viewed as k ICEs, one per fold, and these ICEs

Algorithm 4: Transductive Conformal Evaluator (TCE and *approximate* TCE)

Input: $Z = \{z_0, z_1, \ldots, z_{n-1}\}$, n training examples; $Z^* = \{z_0^*, z_1^*, \ldots\}$, stream of test examples; A, NCM for producing nonconformity scores; $k \in \mathbb{N}$, number of folds—TCE is *approximate* when $k < n$

Output: Stream of boolean decisions $0 = reject, 1 = accept$

Calibration Phase

$\boldsymbol{P} \leftarrow \mathbf{0}$; $i \leftarrow 0$
partition Z equally into $Z^{part} \leftarrow \{ Z'_0, Z'_1, \ldots, Z'_{k-1} \}$
foreach *partition* Z' *of* Z^{part} **do**
  $Z'' \leftarrow Z \setminus Z'$
  $g \leftarrow$ `Fit` (Z'')
  foreach z' *of* Z' **do**
    $\hat{y} \leftarrow g(z')$ ▷ Predicted label
    $Z'_{\hat{y}} \leftarrow \{z \in Z' : z.y = \hat{y}\}$ ▷ Bag of examples with same label
    $\alpha_{z'} \leftarrow A(Z'_{\hat{y}}, z')$ ▷ Nonconformity score
    $S \leftarrow \{A(Z'_{\hat{y}} \setminus \{z\}) : z \in Z'_{\hat{y}}\}$ ▷ Nonconformity scores for bag elements
    $p_{z'} \leftarrow \frac{|\alpha \in S : \alpha >= \alpha_{z'}|}{|S|}$ ▷ Credibility p-value
    $\boldsymbol{P}_i \leftarrow p_{z'}$
    $i \leftarrow i + 1$
  end
end
$\boldsymbol{t}^* \leftarrow$`Transcend.FindThresholds` $(Z, \hat{\boldsymbol{Y}}, \boldsymbol{P})$

Test Phase

$g \leftarrow$ `Fit` (Z)
foreach z^* *of* Z^* **do**
  $\hat{y} \leftarrow g(z^*)$ ▷ Predicted label for test example
  $Z_{\hat{y}} \leftarrow \{z \in Z : z.y = \hat{y}\}$ ▷ Bag of training examples with same label
  $\alpha_{z^*} \leftarrow A(Z_{\hat{y}}, z^*)$ ▷ Nonconformity score
  $S \leftarrow \{A(Z_{\hat{y}} \setminus \{z\}) : z \in Z_{\hat{y}}\}$ ▷ Nonconformity scores for bag elements
  $p_{z^*} \leftarrow \frac{|\alpha \in S : \alpha >= \alpha_{z^*}|}{|S|}$ ▷ Credibility p-value
  if $P_{z^*} < \boldsymbol{t}^*_{\hat{y}}$ **then emit** 0 **else emit** 1
end

operate in parallel to reduce computation time—if the resources are available. However, there is an additional memory cost with storing the separate models.

6.5.4 *Conformal Evaluator Algorithms*

For completeness, we present detailed algorithms for calibrating TCEs, ICEs, and CCEs in Algorithms 4 to 6, respectively.

Algorithm 5: Inductive Conformal Evaluator (ICE)

Input: $Z = \langle z_0, z_1, \ldots, z_{n-1} \rangle$, n training examples; $Z^* = \langle z_0^*, z_1^*, \ldots \rangle$, stream of test examples; A, NCM for producing nonconformity scores; m, number of examples to use for calibration

Output: Stream of boolean decisions $0 = reject, 1 = accept$

Calibration Phase

$\boldsymbol{P} \leftarrow \mathbf{0}$; $\hat{\boldsymbol{Y}} \leftarrow \mathbf{0}$; $i \leftarrow 0$
$Z^{tr} \leftarrow \langle z_0, z_1, \ldots, z_{n-m-1} \rangle$
$Z^{cal} \leftarrow \langle z_{n-m}, z_{n-m+1}, \ldots, z_{n-1} \rangle$
foreach z' *of* Z^{cal} **do**
  $g \leftarrow \texttt{Fit}\ (Z^{cal} \setminus \langle z' \rangle)$
  $\hat{y} \leftarrow \hat{\boldsymbol{Y}}_i \leftarrow g(z')$ ▷ Predicted label
  $Z^{cal}_{\hat{y}} \leftarrow \langle z \in Z^{cal} : z.y = \hat{y} \rangle$ ▷ Bag of examples with same label
  $\alpha_{z'} \leftarrow A(Z^{cal}_{\hat{y}}, z')$ ▷ Nonconformity score
  $S \leftarrow \langle A(Z^{cal}_{\hat{y}} \setminus \langle z \rangle) : z \in Z^{cal}_{\hat{y}} \rangle$ ▷ Nonconformity scores for bag elements
  $p_{z'} \leftarrow \frac{|\alpha \in S : \alpha >= \alpha_{z'}|}{|S|}$ ▷ Credibility p-value
  $\boldsymbol{P}_i \leftarrow p_{z'}$
  $i \leftarrow i + 1$
end
$\boldsymbol{t}^* \leftarrow \texttt{Transcend.FindThresholds}\ (Z, \hat{\boldsymbol{Y}}, \boldsymbol{P})$

Test Phase

$g \leftarrow \texttt{Fit}\ (Z^{tr})$
foreach z^* *of* Z^* **do**
  $\hat{y} \leftarrow g(z^*)$ ▷ Predicted label for test example
  $Z^{cal}_{\hat{y}} \leftarrow \langle z \in Z^{cal} : z.y = \hat{y} \rangle$ ▷ Bag of training examples with same label
  $\alpha_{z^*} \leftarrow A(Z^{cal}_{\hat{y}}, z^*)$ ▷ Nonconformity score
  $S \leftarrow \langle A(Z^{cal}_{\hat{y}} \setminus \langle z \rangle) : z \in Z^{cal}_{\hat{y}} \rangle$ ▷ Nonconformity scores for bag elements
  $p_{z^*} \leftarrow \frac{|\alpha \in S : \alpha >= \alpha_{z^*}|}{|S|}$ ▷ Credibility p-value
  if $P_{z^*} < \boldsymbol{t}^*_{\hat{y}}$ **then emit** 0 **else emit** 1
end

Algorithm 6: Cross-Conformal Evaluator (CCE)

Input: $Z = (z_0, z_1, \ldots, z_{n-1})$, n training examples; $Z^* = (z_0^*, z_1^*, \ldots)$, stream of test examples; A, NCM for producing nonconformity scores; $k \in \{ 2t+1 : t \in \mathbb{N} \}$, number of folds

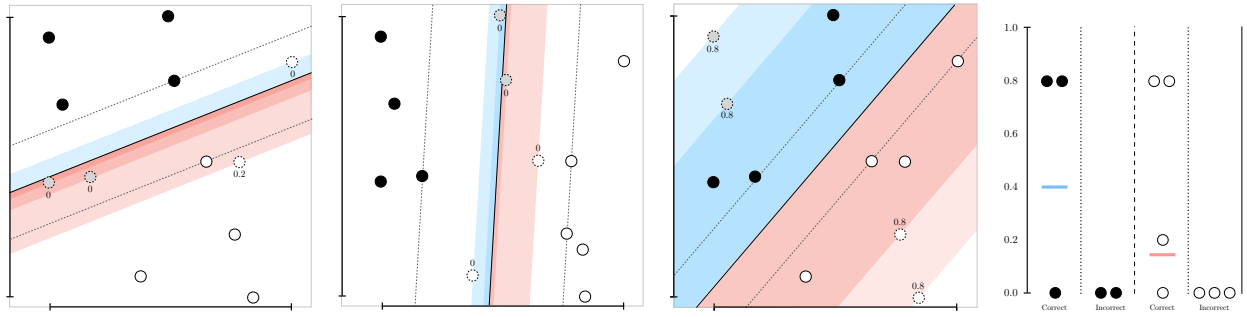
Output: Stream of boolean decisions $0 = reject, 1 = accept$

Calibration Phase

$\boldsymbol{P} \leftarrow \hat{\boldsymbol{Y}} \leftarrow \boldsymbol{G} \leftarrow \boldsymbol{t}^* \leftarrow \boldsymbol{0}$; $i \leftarrow j \leftarrow 0$
partition Z equally into $\{ Z'_0, Z'_1, \ldots, Z'_{k-1} \}$
foreach j *of* $\{ 0, 1, \ldots, k-1 \}$ **do**
  foreach z' *of* Z'_j **do**
    $g \leftarrow$ `Fit` $(Z'_j \setminus (z'))$
    $\hat{y} \leftarrow \hat{\boldsymbol{Y}}_{j,i} \leftarrow g(z')$ ▷ Predicted label
    $Z'_{j\hat{y}} \leftarrow (z \in Z'_j : z.y = \hat{y})$ ▷ Bag of examples with same label
    $\alpha_{z'} \leftarrow A(Z'_{j\hat{y}}, z')$ ▷ Nonconformity score
    $S \leftarrow (A(Z'_{j\hat{y}} \setminus (z)) : z \in Z'_{j\hat{y}})$ ▷ Nonconformity scores for bag elements
    $\boldsymbol{P}_{j,i} \leftarrow \frac{|\alpha \in S : \alpha >= \alpha_{z'}|}{|S|}$ ▷ Credibility p-value
    $i \leftarrow i + 1$
  end
  $\boldsymbol{G}_j \leftarrow$ `Fit` $(Z \setminus Z'_j)$
  $\boldsymbol{T}_j^* \leftarrow$`Transcend.FindThresholds` $(Z'_j, \hat{\boldsymbol{Y}}_j, \boldsymbol{P}_j)$
end

Test Phase

$s \leftarrow 0$
foreach z^* *of* Z^* **do**
  foreach j *of* $\{ 0, 1, \ldots, k-1 \}$ **do**
    $\hat{y} \leftarrow G_j(z^*)$ ▷ Predicted label for test example
    $Z'_{j\hat{y}} \leftarrow (z \in Z'_j : z.y = \hat{y})$ ▷ Bag of training examples with same label
    $\alpha_{z^*} \leftarrow A(Z'_{j\hat{y}}, z^*)$ ▷ Nonconformity score
    $S \leftarrow (A(Z'_{j\hat{y}} \setminus (z)) : z \in Z'_{j\hat{y}})$ ▷ Nonconformity scores for bag elements
    $p_{z^*} \leftarrow \frac{|\alpha \in S : \alpha >= \alpha_{z^*}|}{|S|}$ ▷ Credibility p-value
    if $P_{z^*} \geq \boldsymbol{T}^*_{j\hat{y}}$ **then** $s \leftarrow s + 1$ ▷ Track positive evaluations
  end
  if $s < k/2$ **then emit** 0 **else emit** 1 ▷ Majority vote for final decision
end



(a) First fold (b) Second fold (c) Third fold (d) Alpha assessment

Figure 6.4: Thresholding applied to a linear SVM with approx-TCE (3 folds). P-values, shown above or below each dashed left-out calibration point, are calculated using the negated absolute distance from the decision boundary as an NCM. The shaded regions capture points which are *more nonconform* with respect to the predicted class (blue for class ●, red for class ○). The alpha assessment (d) shows the distribution of p-values and per-class thresholds derived from Q1 of the correctly classified points.

6.6 Sound and Practical Transcendent

Once p-values are calculated, thresholds are derived to decide when to accept or reject new test examples. Here we revise and formalize the strategy used in Transcend [139] and propose a more efficient search strategy.

6.6.1 *Calibration Phase*

The first phase of Transcend [139] is the calibration procedure which searches for a set of per-class *credibility* thresholds $\mathcal{T} = \{\tau_y \in [0,1] : y \in \mathcal{Y}\}$ with which to separate drifting from non-drifting points. Given that low credibility represents a violation of conformal prediction's assumptions, these points are likely to be misclassified by the underlying classifier that similarly relies on the i.i.d. assumption. Note that thresholds can be found with different optimization criteria and it is also possible to threshold on a combination of credibility and *confidence*, which we explore in Section 6.7.3.

Threshold calibration determines how low a credibility should signal drift...

Calibration aims to answer the question: "how low a credibility is too low?", by analyzing the p-value distribution of points in a representative, preferably stationary, environment such as the training set. Which points are selected as calibration points depends on the underlying conformal evaluator, and this comes with various trade-offs (see Section 6.5). Typically, each calibration point (or partition of the calibration set) is held out and the underlying classifier trained on the remaining points. Then a class is predicted for the calibration point(s) with p-values calculated with respect to that predicted class. This process is repeated until all calibration points are assigned a corresponding p-value. Using the ground truth, these p-values can be partitioned into *correct* and *incorrect* predictions that should be separated by $\mathcal{T}$. Methods to find an effective $\mathcal{T}$ can be manual (*e.g.*, picking a quartile visually using an alpha assessment) or automated (*e.g.*, grid search).

Figure 6.4 shows an example of the Transcend [139] thresholding procedure on a toy dataset composed of two classes: ● and ○. A linear SVM is paired with a TCE (Section 6.5) to generate

NCMs and p-values for the binary classification with rejection task. The decision boundary is depicted as a solid line and margins are drawn through support vectors with dotted lines. Due to the use of approximate TCE, the dataset is partitioned into folds, where each fold leaves out four points for calibration and trains on the remainder. The three folds are depicted in Figure 6.4 (a–c). Calibration points are shown with dotted outlines and are faded for class ●.

In each fold, a p-value is calculated for each calibration point as the proportion of other objects that are *at least as dissimilar* to the predicted class as the calibration point itself. In the linear SVM setting shown, less similar objects are those closest to the decision boundary (*i.e.,* those with a higher NCM) residing in the shaded area between the decision boundary and the parallel line intersecting the point (blue for class ● and red for class ○). The calculated p-values are shown aligned above or below each calibration point.

To evaluate how appropriate an NCM is for a given model, the p-values can be analyzed with an *alpha assessment*. Here the distribution of p-values for each class are divided into groups depending on whether the calibration point was correctly or incorrectly predicted as that class. Given that there may not be enough incorrectly classified examples to perform the assessment with, it is standard to perform an alpha assessment in a *non-label-conditional* manner, using p-values computed with respect to all classes, not just each point's predicted class. The greater the margin separating the distributions of correct and incorrect p-values, the better suited an NCM is for a model. The alpha assessment in Figure 6.4 (d) shows the distribution of p-values for correctly and incorrectly predicted calibration points for classes ● and ○. Given the size of the example dataset, the assessment is computed in a *label-conditional* manner and the threshold is set at Q1 of the p-values for correctly classified points (more insight into threshold search strategies can be found in Section 6.6.4). Test points generating p-values below this threshold will be rejected.

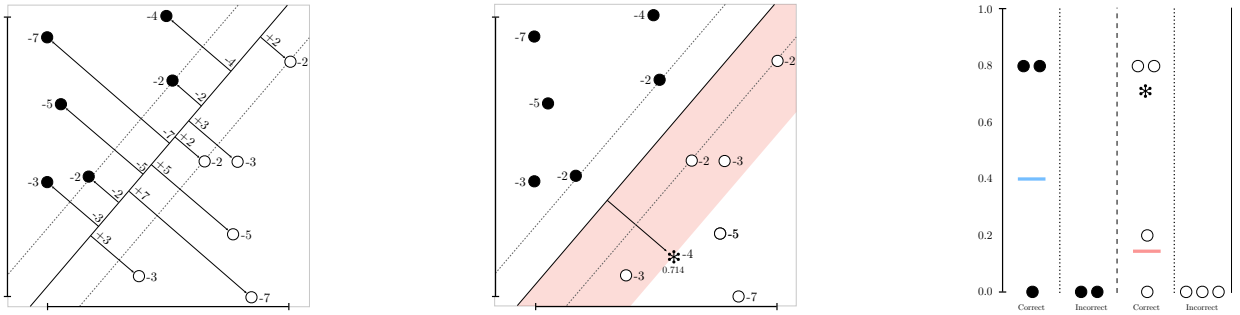
...and tries to separate correct vs. incorrect prediction p-values on calibration data.

6.6.2 *Test Phase*

At test time, there are $|\mathcal{Y}| + 1$ outcomes. When a new testing object z^* arrives, its p-value $p^{\hat{y}}_{z^*}$ is calculated with respect to the predicted class $\hat{y}$ (label conditional). If $p^{\hat{y}}_{z^*} < \tau_{\hat{y}}$, the threshold for the predicted class, then the null hypothesis—that z^* is drifting relative to the training data and does not belong to $\hat{y}$—is approved and the prediction rejected. If $p^{\hat{y}}_{z^*} \geq \tau_{\hat{y}}$, the prediction is accepted and the object classified as $\hat{y}$.

P-values of new test objects are compared to the threshold of the predicted class, points above the threhold are accepted.

Figure 6.5 follows on from the calibration example above. Figure 6.5 (a) illustrates the NCM being used: the negated absolute distance from the hyperplane. In Figure 6.5 (b), a new test example $*$ appears and is classified as class ○. The p-value $p^{\circ}_{*} = 0.714$ is calculated as the proportion of points belonging to ○ with equal or



(a) Distances and NCMs (b) P-value of new point (c) Threshold comparison

Figure 6.5: Test-time procedure applied to a linear SVM and calibrated Transcend [139] with distances from hyperplane and corresponding nonconformity scores shown in (a). In (b) a new test point is classified as class ○. The p-value is calculated as the proportion of points belonging to ○ with equal or greater nonconformity scores than the new point (shaded region). In (c), the new point falls above the threshold for class ○ as derived during the calibration phase (Figure 6.4) and is accepted.

greater nonconformity scores than ∗. Finally, Figure 6.5 (c) shows p°_{*} compared against the threshold $\tau_{\circ}$ and, as $p^{\circ}_{*} \geq \tau_{\circ}$, the prediction is accepted.

6.6.3 *Rejection Cost*

What happens to rejected points depends on the rest of the detection pipeline. In a simple setting, rejected points may be manually inspected and labeled by specialists. Alternatively, they may continue downstream to further automated analyses or to other ML algorithms such as unsupervised systems. In all cases there will be some cost associated with rejecting predictions. When choosing rejection thresholds, it is vital to keep this cost in mind and weigh it against the potential performance gains.

As we outlined in Chapter 5, our Tesseract framework defines three important metrics to use when tuning or evaluating a system for mitigating time decay.

Performance ensures that robustness against concept drift is measured appropriately depending on the end goal (*e.g.*, high F_1 score or high TPR at an acceptable FPR threshold).

Quarantine cost measures the cost incurred by rejections. This is important for putting the performance of kept elements in perspective—there will often be a trade-off between the amount of rejections and higher performance on kept points.

Labeling cost measures the manual effort needed to find ground truth labels for new points. While this is more pertinent to retraining strategies, it is related to the overhead associated with rejection as many may need to be manually labeled. As an example, Miller et al. [194] estimate that the labeling capacity for an average company is 80 samples per day.

Rejection incurs some cost, so finding good thresholds is key...

6.6.4 *Improving the Threshold Search*

Here we model the calibration procedure as an optimization problem for maximizing a given performance metric (*e.g.*, F_1, Precision, or Recall of kept elements). Usually this maximization is subject to some constraint on another metric, for example, it is trivial to attain

Algorithm 7: Transcendent threshold calibration using random search

Input: $\boldsymbol{Y} \in \mathcal{Y}^n$, ground truth labels for n examples; $\hat{\boldsymbol{Y}} \in \mathcal{Y}^n$, predicted labels for n examples; $P \in \mathbb{R}^{n \times |\mathcal{Y}|}$, per-class p-values for n examples

Parameters: $m \in \mathbb{R}$, maximum number of iterations; $\mathcal{F} : \boldsymbol{Y} \times \hat{\boldsymbol{Y}} \times P \longrightarrow \mathbb{R}$, performance measure to optimize (*e.g.*, F_1); $\mathcal{G} : \boldsymbol{Y} \times \hat{\boldsymbol{Y}} \times P \longrightarrow \mathbb{R}$, performance measure to constrain (*e.g.*, kept examples); $\mathcal{C} \in \mathbb{R}$, lower bound for constrained measure $\mathcal{G}$

Output: $\boldsymbol{t}^* \in [0,1]^{|\mathcal{Y}|}$, a vector of per-class thresholds

```
t* ← 0 ; counter ← 0
while counter < m do
    t ←$ [0,1]^|Y|                                   ▷ Pick random thresholds
    if F(Y, Ŷ, P; t) > F(Y, Ŷ, P; t*) and G(Y, Ŷ, P; t) ≥ C then
        t* ← t
    else if F(Y, Ŷ, P; t) = F(Y, Ŷ, P; t*) and G(Y, Ŷ, P; t) > G(Y, Ŷ, P; t*) then
        t* ← t
    counter ← counter + 1
end
return t*
```

high F_1 performance in kept elements by accepting very few high quality predictions, but this will cause many correct predictions to be rejected.

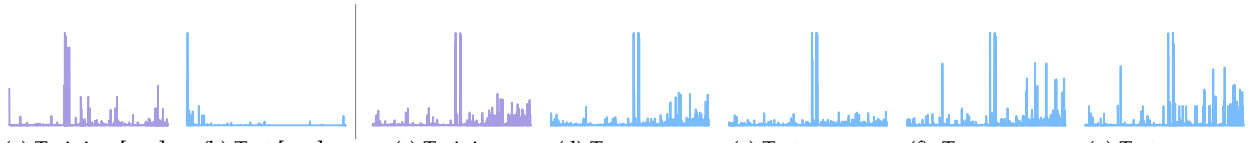
Formally, given n calibration points, we represent this as:

$$\begin{aligned} \arg\max_{\mathcal{T}} \quad & \mathcal{F}(Y, \hat{Y}, P; \mathcal{T}) \\ \text{subject to} \quad & \mathcal{G}(Y, \hat{Y}, P; \mathcal{T}) \geq \mathcal{C}\,, \end{aligned} \tag{6.6}$$

where $\boldsymbol{Y}$ and $\hat{\boldsymbol{Y}}$ are n-dimensional vectors of ground truth and predicted labels respectively, P is a $|\mathcal{Y}| \times n$-dimensional matrix of calibration p-values and $\mathcal{T} = \{\, \tau_y \in [0,1] \mid y \in \mathcal{Y} \,\}$ is the set of thresholds. The objective function $\mathcal{F}$ maps these inputs to the metric of interest in $\mathbb{R}$, for example F_1 of kept elements, while $\mathcal{G}$ maps to the metric to be constrained, such as the number of per-class rejected elements. $\mathcal{C}$ is the threshold value that bounds the constraint function.

...so we propose a random search strategy to improve on the expensive grid search of the original.

Given this formalization, we propose an alternative random search strategy to replace the exhaustive grid search used in the original paper [139]. In the exhaustive grid search, each possible combination of thresholds over all classes is tested systematically, considering some fixed range of variables $V = \{v : v \in [0,1]\}$. However, this suffers from the curse of dimensionality [31], resulting in $|V|^{|\mathcal{Y}|}$ total trials, growing exponentially with the number of classes. Additionally, reducing the granularity for V increases the risk of 'skipping' over an optimal threshold combination. Similarly, often many useless threshold combinations are considered (where one is either too high or too low). This failure to evenly cover subspaces of interest worsens as the dimensionality increases [33], making it especially problematic for multiclass classification. The granularity at values are chosen for V can be chosen manually based on intuition and past experience, however this results in experiment parameters which are difficult to reproduce and transfer to other settings.



(a) Training [139] (b) Test [139] (c) Training (d) Test, 1 year (e) Test, 2 years (f) Test, 3 years (g) Test, 4 years

Figure 6.6: Frequency distributions of features depicting covariate shift between training and test malware examples. The original data [139] displayed in (a) and (b), shows a sudden and significant shift, while our data, displayed in (c–g), shows a more subtle drift occurring over time.

It has been shown for hyperparameter optimization that random search is able to find combinations of variables at least as optimal as those found with full grid search over the same domain, at a fraction of the computational cost [33]. We apply these findings to the threshold calibration and replace the exhaustive grid search with a random search (Algorithm 7). We choose random combinations of thresholds in the interval $[0, 1]$, keeping track of the thresholds that maximize our chosen metric given the constraints (see Section 6.6.4). The search continues until either of two conditions are met. A limit is set on the number of iterations, determined by the time and resources that are available for the calibration. Intuitively a higher limit will increase the likelihood of finding better thresholds and so acts as the upper bound of the optimization. Secondly, a stop condition can be set. In this work we consider a *no-update* approach in which the search will stop once a fixed point is found, *i.e.,* if there is no improvement to performance after a certain number of consecutive iterations. Note that this search can be parallelized. We empirically compare the two search strategies in Section 6.7.4.

6.7 Experimental Evaluation

We evaluate our novel evaluators when faced with a gradual concept drift caused by the evolution of real malicious Android apps over time (Section 6.7.2), the performance gained by including confidence scores (Section 6.7.3), how our random search implementation fares against exhaustive search (Section 6.7.4), how the evaluators compare to alternative methods (Section 6.7.5), and perform on PE and PDF malware domains (Section 6.7.6).

6.7.1 *Experimental Settings*

To evaluate, we build Transcendent, encompassing the original functionality and our new proposals...

Prototype We develop a prototype of Transcendent that encompasses the functionality of the original work, Transcend [139], as well as our new proposals. The prototype is implemented as a Python library that aims to be familiar to users of popular ML frameworks such as scikit-learn [221]. We release the code as open source and make it available to other researchers (details in the front matter). Note that this is the first publicly available implementation of Transcend [139] in any form.

Dataset We first focus on malware detection in the Android domain. We sample 232,848 benign and 26,387 malicious apps from AndroZoo [8]. This allows us to demonstrate efficacy when faced with a surreptitious concept drift typical to mobile malware. The apps span 5 years, from Jan 2014 through to Dec 2018. We use the Tesseract framework to carry out temporal evaluations, ensuring that Tesseract's constraints are accounted for to remove sources of spatial and temporal experimental bias. Training and calibration are performed using apps from 2014 and testing is evaluated over the concept drift that occurs over the remaining period on a month-by-month basis.

...applied to a 5 year dataset of Android malware containing 232,848 goodware and 26,387 malware...

Eliminating Sampling Bias The original evaluation of Transcend [139] artificially simulated concept drift by fusing two datasets: Drebin [19] and Marvin [174], a process which may have induced experimental bias [20] and made it easier to detect drifting examples. Figure 6.6 shows a visibly significant covariate shift in the distribution of features for training and test malware examples from Jordaney et al. [139], with a Kullback-Leibler (KL) divergence [157]—an unbounded measure of distribution difference—of 696.66. The covariate shift in our dataset is much more subtle over time, with an average KL divergence of 189.55 between each training and test partition. From this we conclude that the distributions were significantly more different in the original evaluation than would be expected in realistically occurring concept drift, which would have made it easier to detect drifting examples.

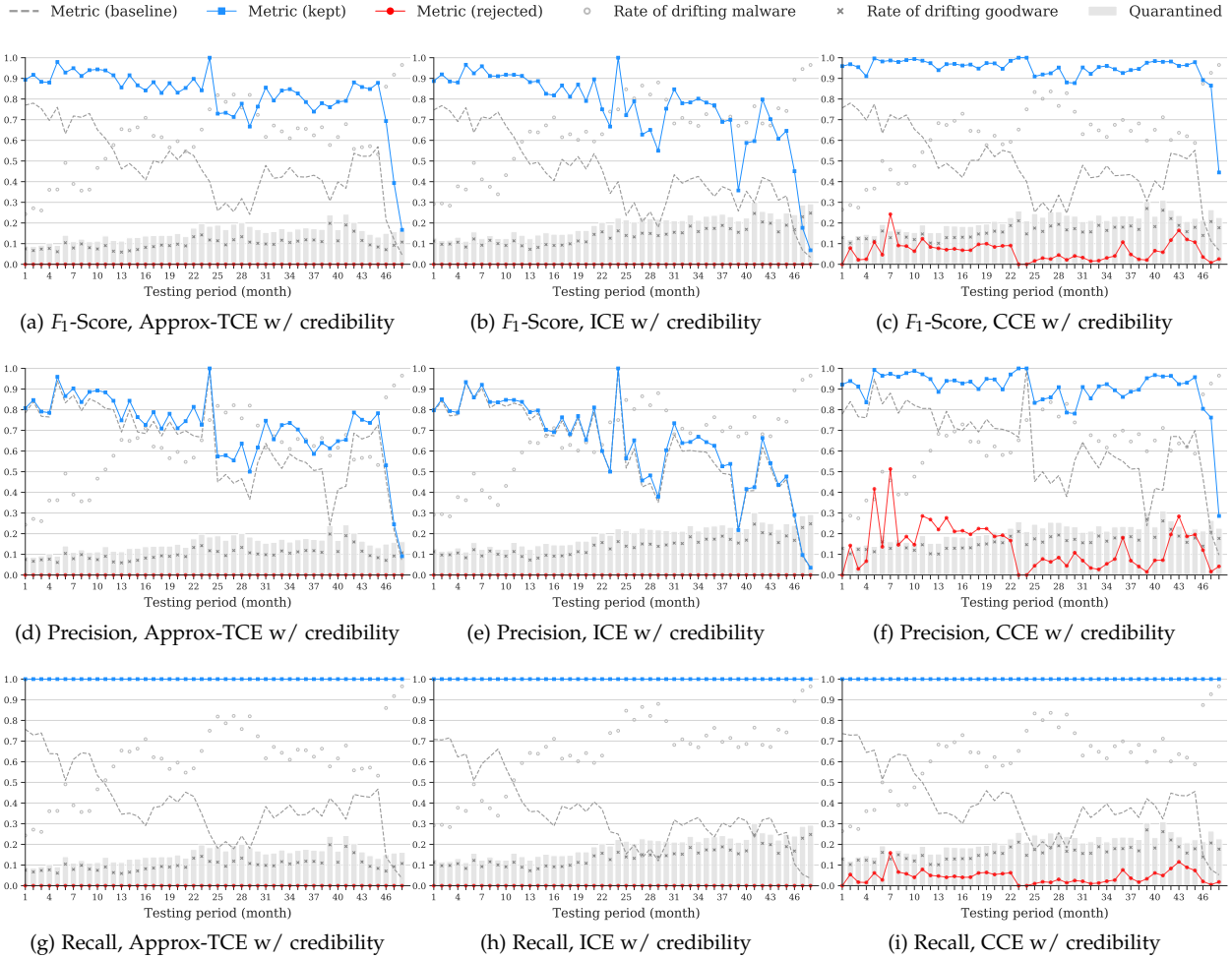
...containing realistic drift patterns...

Classifier For the underlying classifier, we use Drebin [19] which we earlier showed can achieve state-of-the-art performance if a retraining strategy is used to remediate concept drift (see Chapter 5). Due to this, we hypothesize that if Transcend [139] is used to reject drifting points, Drebin will be able to classify the remaining points with high accuracy. Drebin uses a linear SVM and a binary feature space where components (activities, permissions, URLs, etc) are represented as present or absent.

...with Drebin as the underlying classifier...

Calibration To optimize the thresholding, we maximize the F_1 of all kept elements for a rejection rate less than 15%. These metrics are computed in aggregate for each time period of the temporal evaluation. On our dataset, this would amount to an average rejection of ~20 samples a day, well below the estimated labeling capacity of 80 a day suggested by Miller et al. [194]. However, we note that these constraints may need to be adjusted according to specific operational requirements, for example, it may be more appropriate to minimize the rejection rate while sacrificing F_1 for kept elements. For the random search we use 100,000 random iterations with early stopping after 3,000 consecutive events without improvement. For approx-TCE and CCE we calibrate using $k = 10$ folds.

...and calibrated to maximize F_1-Score given a reasonable rejection rate.



(a) F_1-Score, Approx-TCE w/ credibility (b) F_1-Score, ICE w/ credibility (c) F_1-Score, CCE w/ credibility

(d) Precision, Approx-TCE w/ credibility (e) Precision, ICE w/ credibility (f) Precision, CCE w/ credibility

(g) Recall, Approx-TCE w/ credibility (h) Recall, ICE w/ credibility (i) Recall, CCE w/ credibility

Figure 6.7: Performance for the three proposed conformal evaluators (columns) without rejection (dashed gray), and for accepted (□-blue) and rejected (○-red) examples. Bars show the proportion of rejected elements each period, while the x and o markers show the proportion of *ground truth* malware and goodware identified as drifting, respectively.

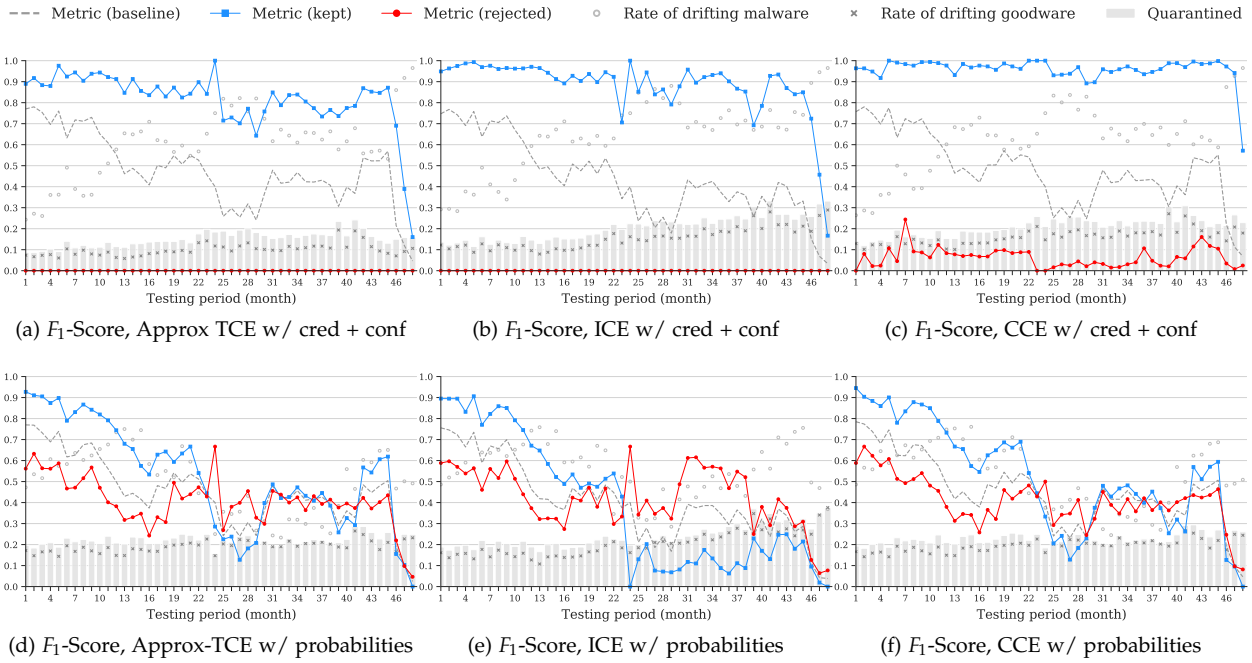
6.7.2 *Novel Conformal Evaluators*

We compare the performance of our new evaluators...

Here we compare the novel conformal evaluators of Transcendent. As vanilla TCE is not feasible for this experiment setting due to the size of the training set (Section 6.5), we use approx-TCE as a stand-in, however we provide a minimal experiment in Section 6.7.7 to show that the expected performance difference between vanilla TCE and our evaluators is negligible.

Performance Metrics Figure 6.7 shows the the F_1, Precision, and Recall (rows 1–3) for each of the novel evaluators (columns). The middle dashed line shows the baseline performance when no rejection is enforced. This is the performance decay caused by concept drift present in the dataset that results from an evolving malicious class. Note that classifiers degrade rapidly, becoming worse than random in under one year.

The upper solid line shows the performance of kept elements, those with test p-values that fall above the threshold of their predicted classes. While decay is still present, approx-TCE and ICE are able to maintain $F_1 > 0.8$ for two years, doubling the lifespan of the model. Note that the sudden drop in performance of the last three months is likely an artifact of the fewer examples crawled by



(a) F_1-Score, Approx TCE w/ cred + conf (b) F_1-Score, ICE w/ cred + conf (c) F_1-Score, CCE w/ cred + conf

(d) F_1-Score, Approx-TCE w/ probabilities (e) F_1-Score, ICE w/ probabilities (f) F_1-Score, CCE w/ probabilities

Figure 6.8: Performance for the three proposed conformal evaluators (columns) using alternative quality metrics, without rejection (dashed gray), and for accepted (□-blue) and rejected (○-red) examples. Bars show the proportion of rejected elements each period, while the x and o markers show the proportion of *ground truth* malware and goodware identified as drifting, respectively.

AndroZoo in those months.

The lower solid line shows the performance of rejected elements. Low metrics mean the rejected elements would have been incorrectly classified by the underlying classifier and were rightfully rejected, while high metrics means rejections were erroneous. Approx-TCE and ICE have F_1, Precision, and Recall of 0 for rejected elements for every test month meaning that all rejected elements would have been misclassified.

The result of CCE differs in that it is less conservative in its rejections. The performance of kept elements is much higher, but also slightly higher for rejected elements, indicating that a small proportion of rejected elements would have actually been correctly classified. We observe that this conservatism can be increased or decreased by modifying the conditions of the majority vote. If more folds are required to agree before a decision is accepted, the CCE will be more conservative, rejecting more elements. If less folds are required, more elements will be accepted. In this respect, CCE offers an alternative dimension of tuning in addition to the threshold optimization. Additionally, this is parameter can be altered during a deployment, rather than being set at calibration. This allows for some adaptability, such as when the cost of False Negatives is very high (*e.g.*, not alerting security teams to attacks in network intrusion detection), or when the cost of False Positives is very high (*e.g.*, withholding benign emails in spam detection, or disabling legitimate user accounts in fake account detection). A further empirical analysis of the effect of the majority vote conditions is included in Section 6.8.1

...showing that they can more than double the lifetime of the model...

Rejection Rates Gray bars show the proportion of rejected test elements. In each case, rejections begin close to the rate used for

		Approx-TCE	ICE	CCE
Baseline	AUT(F_1 w/ credibility, 48m)	.480	.440	.483
	AUT(F_1 w/ cred + conf, 48m)	.480	.440	.483
	AUT(F_1 w/ probability, 48m)	.456	.405	.455
Kept Elements	AUT(F_1 w/ credibility, 48m)	.829	.762	.950
	AUT(F_1 w/ cred + conf, 48m)	.822	.887	.962
	AUT(F_1 w/ probability, 48m)	.531	.388	.532
Rejected Elements	AUT(F_1 w/ credibility, 48m)	.000	.000	.064
	AUT(F_1 w/ cred + conf, 48m)	.000	.000	.063
	AUT(F_1 w/ probability, 48m)	.410	.426	.410

Table 6.2: AUT using different quality metrics: credibility, credibility with confidence, and probabilities (cf. Figures 6.7 and 6.8). We aim to *maximize* the metrics of kept elements and *minimize* the metrics for rejected elements.

calibration before slowly rising each year, averaging 26.45 samples per day. While rejection rates may appear high, these are symptomatic of rising concept drift and deteriorating performance in the underlying classifier and are often preferable to taking incorrect actions on False Positives and False Negatives. In an extreme case where a classifier always predicts the incorrect label, rejection rates could reach 100% but the F_1 of rejected elements would be 0%. The gray markers show the proportion of ground truth malware and goodware that are rejected each period, illustrating the evaluators' perception of drift in that class. Strikingly, for our evaluators the drift rate of the malicious class is *inversely correlated* to the performance loss in the baseline, while the drift rate for goodware is relatively stable. This supports our hypothesis that performance decay is mostly driven by evolution in the malicious class. We reiterate that in the case of Approx-TCE and ICE, the low F_1 of rejected elements indicates that *all* of the rejected malware would have evaded the classifier if they had not been identified as drifting.

...and demonstrating how drift is driven by the malicious class...

Runtime The runtime of the conformal evaluators during this experiment match what would be expected from their relative complexities (cf. Table 6.1). The ICE is the quickest at 11.5 CPU hours. The CCE took 35.6 CPU hours, but our implementation is parallelized resulting in a wall-clock time identical to the ICE. The Approx-TCE took 46.1 CPU hours. As discussed, vanilla TCE was computationally infeasible, but we estimate a runtime of 1.9 CPU years, considering that the time required to fit the underlying classifier once is ~10 minutes and the classifier must be trained once *for each* training example. We conclude that the ICE is the most useful for settings where resources are limited or models with a rapid iteration cycle (*e.g.,* daily), while the CCE offers greater confidence and flexibility at a slightly higher computational cost.

...while drastically reducing the runtime of the original.

We conclude that the ICE is the most useful for settings where resources are limited or models with a rapid iteration cycle (*e.g.,* daily), while the CCE offers greater confidence and flexibility at a slightly higher computational cost.

6.7.3 *Credibility, Confidence, and Probabilities*

Here we compare the performance under different quality metrics. The exact performance over time for all settings discussed in this subsection is reported in Table 6.2.

We compare our evaluators using different quality metrics other than credibility alone...

Credibility with Confidence Intuitively, including confidence thresholds when evaluating a classifier prediction would be beneficial as confidence represents how certain the classifier is in its own prediction. However, as credibility is the main indicator that i.i.d. has been violated, and thus that concept drift is occurring, it is unclear what further gain confidence could provide. Here we test this by evaluating the conformal evaluators under the same conditions as Section 6.7.2, using per-class thresholds for both credibility and confidence.

Figure 6.8 compares the F_1 for each conformal evaluator (columns) using alternative thresholding metrics (compared to row 1 of Figure 6.7). The upper blue line shows the performance of kept elements while the lower red line shows the performance of rejected elements. The gray dashed line depicts the baseline performance when no rejection mechanism is used.

...showing that confidence can stabilize results at a higher cost...

The first row shows the F_1 when confidence is included. Performance for the approx-TCE and CCE is relatively unchanged, but is markedly improved for the ICE with degradation postponed much longer than before. The confidence appears to restore some of the statistical information lost by using only a small amount of the training data for calibration.

However, the computation required to find thresholds is higher than with credibility only—equivalent to doubling the number of classes. We conclude that the performance gain from including confidence is situationally dependent; although it will improve the accuracy of an ICE, a CCE will often provide the same accuracy with comparable calibration time.

...and that simple probabilities are ineffective for thresholding.

Probabilities The latter row of Figure 6.8 shows the F_1 when the classifiers' output probabilities are used for thresholding, rather than generating per-class p-values for each calibration and test point. For each evaluator, the same training and calibration split is used as with p-values, to ensure a fair comparison. The plot shows probabilities alone offer a very small improvement for kept elements over the baseline in the first year, becoming increasingly volatile as the concept drift becomes more severe. Additionally, the perceived drift rate for each class has no relation to the baseline performance loss, indicating that the root cause of the drift is not identified. This shows the statistical support offered by the conformal evaluator's p-value computation is significant and justifies the additional computational overhead that it induces.

	FPs	FNs	Prec.	Rec.	F_1	#Trials
No rejection	3,529	19,486	0.98	0.92	0.95	N/A
Full grid	2,187	0	0.99	1.00	0.99	1,317,520
Random	3,259	0	0.98	1.00	0.99	10,000

Table 6.3: Performance of optimal thresholds discovered using a full grid search vs. random search. Random search discovers thresholds equivalent to the full grid search but with two orders of magnitude fewer trials (Section 6.7.4).

6.7.4 *Full Grid Search vs. Random Search*

Here we evaluate our random search implementation (Section 6.6.4) compared to the full grid search used in the original proposal [139]. We show the random search can find high quality calibration thresholds more efficiently than the full search.

Due to the full grid search cost, here we train and calibrate on 1 month of data and test on the remaining 59 months using an approx-TCE with 10 folds. We maximize F_1 for an acceptable rejection rate of less than 15%. To ensure the baseline discovers high quality thresholds we use a fine granularity grid covering 1,317,520 combinations of thresholds. For random search we set an upper limit of 10,000 trials.

Our new threshold search reduces the cost of finding thresholds by two orders of magnitude.

Table 6.3 compares the performance without rejection, with rejection thresholds from the full grid search, and with rejection thresholds from random search. Note there is no significant performance difference between the two strategies, but the random search is able to cover the same search space with two orders of magnitude fewer trials. We observe that the full grid search makes erroneous assumptions on the distribution of quality thresholds which the random search does not. Additionally, while the random search allows for a variety of stopping conditions, the only mechanism to control the length of the full grid search is the size of the interval to search and the granularity of the search steps—which are difficult to choose beforehand.

We compare against two prior approaches with mechanisms similar to Transcendent...

6.7.5 *Comparison to Prior Approaches*

To provide further context on the performance of Transcendent, we compare against two closely related state-of-the-art approaches: Linusson et al. [175] (which we denote CP-Reject) and DroidEvolver [316].

CP-Reject [175] The first approach is a recent method for performing rejection using conformal prediction. For a given test set and hyperparameter k, CP-Reject aims to output the largest possible set of predictions containing on average no more than k errors, while rejecting test objects for which it is too uncertain. The training and calibration dataset splits are the same as we use for the ICE setting; however while Transcendent makes decisions on individual test objects as they appear, CP-Reject operates *a posteriori* on a batch of test inputs and predictions. Given this advantage, to ensure a fair comparison we test on each month with k set to the 85th percentile which ensures a rejection rate of 15%—the same rejection

...CP-Reject [175] which also builds on conformal prediction...

rate Transcendent is calibrated for. The underlying classifier is a random forest classifier with 100 trees and the conformal prediction NCM is the maximum margin between the output probability for the predicted class and the output probabilities for all other classes.

DroidEvolver [316] The second approach is a state-of-the-art Android malware detector designed for drift *adaptation*, but that includes a rejection component, in which the drift identification mechanism is inspired by the original Transcend [139]. DroidEvolver is built on an ensemble of five linear online learners, with a weighted sum as the ensemble decision function. For each new test object a *juvenilization indicator* (JI) score is computed per model as the proportion of apps in a fixed-size buffer of previously encountered apps, of the same class, that have decision scores greater than the new object. An object is marked as drifting when the JI score falls outside of a precalibrated range and the corresponding decisions are rejected, *i.e.*, excluded from the weighted sum which is used to pseudo-label and update with the drifting point. The ongoing performance of the system relies on the quality of the pseudo-labels and thus indirectly on the quality of the drift identification. The JI scores are very similar to the credibility p-values from conformal evaluation, with the computational complexity of full TCE being addressed by using the small fixed-size app buffer: drift identification should be effective so long as the app buffer is representative of the overall data population. Due to this relationship, it is informative to compare against Transcendent.

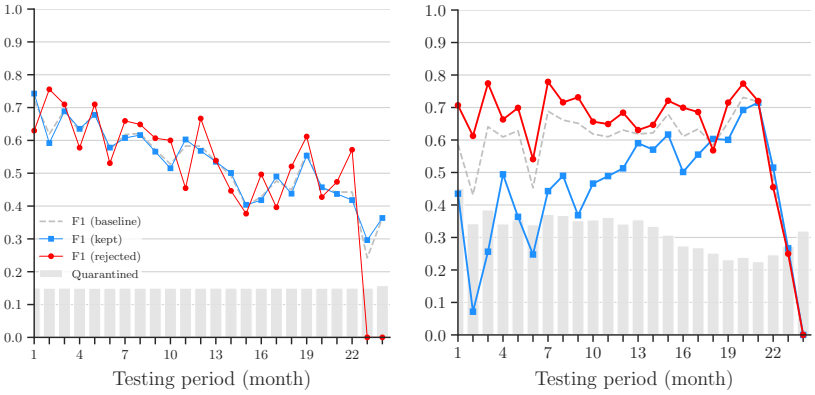
...and DroidEvolver [316] which uses an ensemble of online learners to adapt to drift...

Results Figure 6.9 shows the F_1 performance of CP-Reject and DroidEvolver trained and calibrated on the first year of the dataset and tested on the two subsequent years at monthly intervals. These can both be compared to the first 24 months of ICE and CCE results of Figure 6.7 (b–c).

For CP-Reject, the similar F_1 performance for kept, rejected, and baseline predictions indicate that it is unable to distinguish between drifting and non-drifting points. Although it may effectively reject low-quality predictions in a stationary environment, conformal prediction relies heavily on the exchangeability assumption, which is violated in this dataset. To obtain a prediction, CP-Reject follows conformal prediction principles and outputs the class with the highest credibility (see Section 6.4.3), but we argue this output is not trustworthy under drifting conditions. Conversely in conformal *evaluation*, by decoupling the prediction of the underlying classifier from the rejection mechanism and directly interpreting the credibility as a measurement of drift when comparing it to the calibrated thresholds, we can more effectively detect poor quality predictions.

...but neither are as successful at identifying drifting examples.

While the detection performance of DroidEvolver is mediocre on this dataset, the pseudo-labeling update mechanism manages to stabilize the system against the impact of drift up until the last four months. After this, performance deteriorates due to the poor



(a) CP-Reject [175]

(b) DroidEvolver [316]

Figure 6.9: F_1-Score over time for two prior approaches with mechanisms similar to Transcendent (cf. Figure 6.7 (b–c)).

quality of pseudo-labels used for updating the online models—as DroidEvolver uses predicted labels as pseudo-labels, the negative feedback loop is difficult to recover from. Surprisingly, the drift identification mechanism rejects more correct predictions than it keeps for each test period. We posit that the small app buffer fails to sufficiently represent the true app population, which may in turn lead to the negative feedback loop in the later months. Although much more extreme here, this informational inefficiency is also responsible for the variability we see when using ICEs—different dataset splits may be more or less representative of the true distribution and result in better or worse accuracy, a phenomenon that is mitigated by using a CCE.

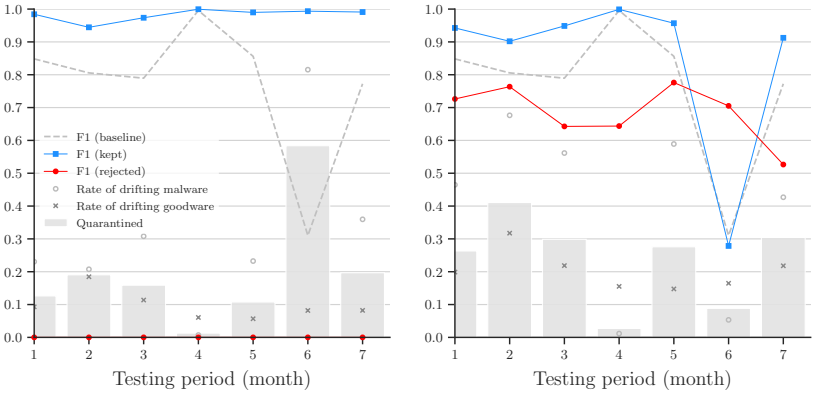
6.7.6 *Beyond Android Malware and SVMs*

We also test on PDF other malware domains and classifiers...

While Transcendent and conformal evaluation are agnostic to the underlying classifier and feature space, we have so far focused on detecting Android malware with a linear SVM. Here we demonstrate the performance beyond this setting. To simplify the axes of comparison, we apply an ICE to each setting, using credibility p-values and random search for threshold calibration with the same constraints as before.

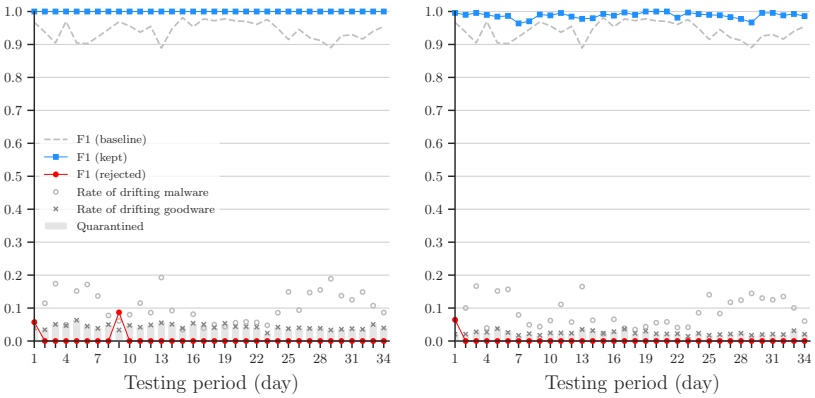
...Windows PE malware using a GBDT...

Windows PE malware with GBDT We take examples from the EMBER v2 dataset [11] spanning 2017, containing 47,888 benign and 69,202 malicious executables (labeled as having 40+ VirusTotal AV detections). The feature space contains a diverse set of features which can be categorized as either parsed features (*e.g.*, header information), histograms (*e.g.*, byte-value histograms), and printable strings (*e.g.*, URL frequency). As the underlying classifier, we use gradient boosted decision trees (GBDT) [106] as in Anderson and Roth [11], and for the NCM we use the output probability for the predicted class, negated for positive predictions. We train on executables from the first five months and test on the remaining.



(a) Windows PE, ICE, credibility

(b) Windows PE, ICE, probabilities



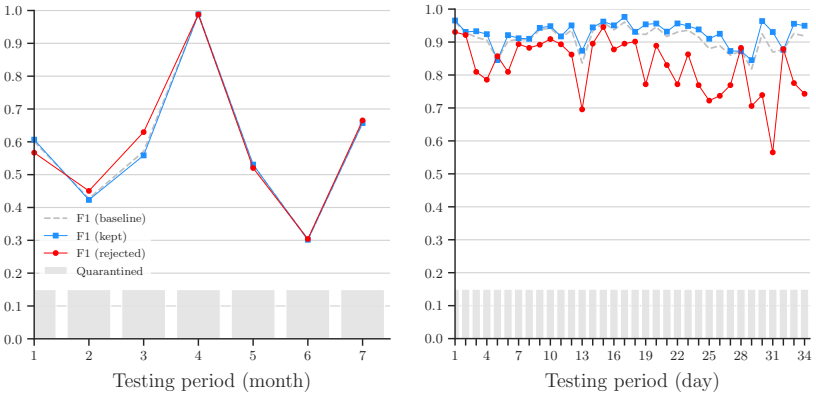
(c) PDF, ICE, credibility

(d) PDF, ICE, probabilities

Figure 6.10: F_1-Scores for EMBER Windows PE malware [11] using GBDT (top row) and Hidost PDF malware [272] using RBF SVM (bottom row).

PDF malware with RF We use examples from the Hidost dataset [272] spanning five weeks in Aug–Sep 2012, consisting of 181,792 benign and 7,163 malicious files (labeled as having 5+ VirusTotal AV detections). The feature space is created by statically parsing the PDF files to extract structural paths in the PDF hierarchy that map to boolean or numeric feature values, such as the presence of certain PDF objects or metadata such as the number of pages. As the underlying classifier we use a random forest (RF) classifier following Srndic and Laskov [272]. As the NCM we use the proportion of decision trees that disagree with the prediction of the ensemble (as illustrated in Figure 6.1 (e)). Interestingly, a major contribution of the Hidost feature space in contrast to prior approaches [*e.g.*, 271] is that similar features are consolidated in order to be *more robust to drift*. This means the distribution should be relatively stationary compared to the Android dataset and will allow us to test whether Transcendent is able to make effective decisions on prediction quality when drift is less severe.

Note that we are unable to find authoritative measurements for the expected class balance for PE and PDF malware in the wild as we are for Android malware (see Section 5.3.3), so we defer to the class balance in the original datasets. This may result in a slight spatial bias if the class balance is unrealistic, however all approaches will be affected equally. Additionally, we can here examine whether the class balance affects the ability for Transcendent



(a) EMBER Windows PE malware [11]

(b) Hidost PDF malware [272]

Figure 6.11: F_1-Score of CP-Reject [175] on alternative malware datasets.

to identify low quality predictions.

Results The results for Windows PE malware (Figure 6.10 (a–b)) are consistent with those on Android data. Transcendent outperforms probabilities alone which tend to reject many otherwise correct predictions. In particular, a large spike in drifting malware affects month six which probabilities are unable to cope with, while Transcendent raises the rate of rejections accordingly without making any additional errors.

...and show that despite a different severity of drift in each dataset...

As noted earlier, the PDF dataset gives us the opportunity to evaluate Transcendent on a relatively stationary distribution (Figure 6.10 (c–d)). As expected, thresholding using probabilities is much more effective than it is in a drifting setting, however it under-rejects compared to Transcendent, which is able to find thresholds that push the F_1-Score of kept predictions to 1.0 while rejecting almost entirely incorrect predictions. Exceptions to this are months one and nine, in which a small quantity of true positive predictions are rejected. However this is anomalous (*i.e.*, it does not continue as drift increases) and could be mitigated by calibrating with a constraint on the F_1 of rejected samples rather than the F_1 of kept examples alone. From this we conclude that Transcendent is useful for maximizing the potential of a high-quality robust classifier, and does not rely on relatively severe drift—as present in the Android dataset—to detect low quality decisions. To this end Transcendent can be combined with robust feature spaces which is an orthogonal direction to combating concept drift.

...Transcendent still improves the quality of accepted predictions...

As an additional result we show in Figure 6.11 that Transcendent outperforms CP-Reject for both domains (we exclude DroidEvolver as it is specific to Android malware). Similar to the results on the Android dataset (Section 6.7.5), the overall ability for CP-Reject to distinguish between drifting and non-drifting points is poor on the PE malware dataset. For the PDF malware dataset, which exhibits much less drift, CP-Reject is significantly more effective, which supports the hypothesis that it is the violation of conformal prediction's exchangeability assumption which results in the lower performance on the Android and PE datasets. Nevertheless,

...and outperforms CP-Reject in both domains.

	TCE	Approx-TCE	ICE	CCE
Baseline	0.68	0.70	0.45	0.69
Kept Elements	0.97	0.97	0.94	1.00
Rejected Elements	0.00	0.00	0.00	0.21

Table 6.4: AUT(F_1, $7m$) comparing vanilla TCE to our novel conformal evaluators on Windows PE malware data. To be computationally viable, 10% of the training data was randomly sampled to use for training and calibration.

Transcendent with credibility (and even probabilities) outperforms CP-Reject in this setting also (cf. Figure 6.10 (c)).

6.7.7 *Full Vanilla TCE on EMBER Subset*

A full scale comparison to the original TCE is not possible due to its computational complexity—recall that one classifier must be trained for each example in the training set. However, it is informative to perform a small-scale experiment as there may be settings where the vanilla TCE *is* viable, and we wish to ensure that there is no significant performance difference between vanilla TCE and our novel conformal evaluators.

Given the intractability of running a full-scale TCE experiment...

We perform an experiment on the Windows PE malware dataset, where 10% of the training data is randomly sampled to use for training and calibrating the evaluators (this is the largest subsample we can take given our resource constraints). We choose the PE dataset over the Android dataset due to the high dimensionality of the Android feature space that may cause instability when the number of examples is very low, and over the PDF dataset which is relatively stationary and may make it harder to discern performance differences between the different evaluators. One caveat of this subsampling is the reduced performance of the baseline for the ICE, which is due to the reduced data available to the proper training set, although Transcendent appears unaffected by this.

...we compare against TCE on the Windows PE dataset utilizing only 10% of the training split...

Table 6.4 summarizes the F_1 performance over the seven month-long test periods using the area-under-time (AUT) metric. The performance difference between TCE and our evaluators in terms of distinguishing between drifting and non-drifting examples is negligible, shown by the very high AUT of kept elements and very low AUT of rejected elements. That is, there is little to no performance sacrifice when using our evaluators over the vanilla TCE. The overall trends otherwise follow those in our main Android experiments (cf. Figure 6.7).

...showing comparable or improved results for all our newly proposed conformal evaluators.

6.8 Operational Considerations

Here we discuss some actionable points regarding the use of conformal evaluation and Transcendent.

Transcendent in a Detection Pipeline Transcendent has particular applications in detection tasks where there is a high cost of False Positives, (*e.g.*, spam [206], malware [271, 19, 70], fake accounts [47, 42]).

In these cases, it may be preferable to avoid taking a decision on low-confidence predictions or, where a gradated response is possible, diverting rejected examples towards alternative remediation actions. Consider an example in the fake accounts setting: owners of accounts in the set of rejected positive predictions can be asked to solve a CAPTCHA on their next login (a relatively easy check to pass) while the owners of accounts in the set of kept positive predictions can be asked to submit proof of their identity. Increasing rejection rates signal a performance degradation of the underlying classifier without immediately submitting to the errors it produces, giving engineers more time to iterate and remediate.

Transcendent is particularly useful in detection tasks with a high cost of False Positives...

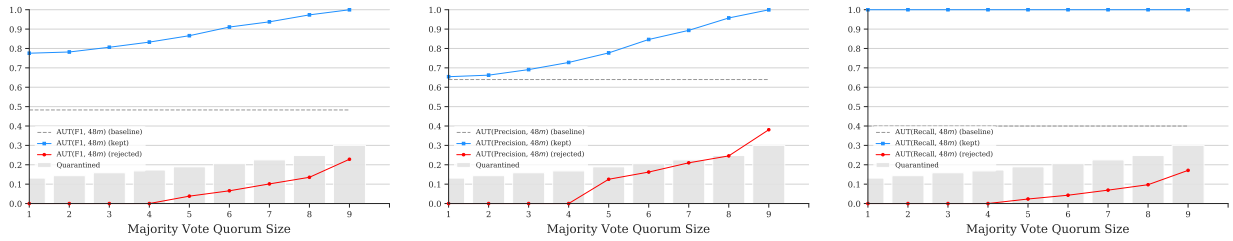
Operational Recommendations Based on our empirical evaluation (Section 6.7), we make the following recommendations for Transcendent deployments:

- Transcendent is agnostic to the underlying learning algorithm, but the quality of the rejection relies on the suitability of the NCM. Some examples of possible NCMs for different types of classifiers are described in Figure 6.1.
- Using an ICE or CCE is preferred over TCE due to their computational efficiency, and are preferred over approx-TCE due to approx-TCE's reliance on assumptions that may not universally hold.
- ICEs are relatively fast and lightweight and excel when resources are limited. CCEs make rejections with higher confidence but at a higher computational cost.
- Thresholding with credibility alone is sufficient to achieve high quality prediction across all conformal evaluators. While confidence can improve the stability of an ICE (Section 6.7.3), it requires greater calibration time.
- Random search is preferred over exhaustive grid search as it finds similarly effective thresholds at a significantly lower cost.
- Rising rejection rates should be interpreted as a signal that the underlying model is degrading. This signal can be used to trigger model retraining or other remediation strategies.

...and we give some guidance for such deployments.

6.8.1 *Analysis of CCE Tuning*

As a further empirical analysis of the effect of the majority vote conditions on performance, here we explore how the size of the quorum for CCE affects the rejection decision. Figure 6.12 shows the performance over time summarized using the area-under-time (AUT) metric (see Section 5.5.2) for F_1 (a), Precision (b), and Recall (c). Note that Figure 6.12 omits the setting where the majority vote



(a) AUT(F_1, 48m) (b) AUT(Precision, 48m) (c) AUT(Recall, 48m)

Figure 6.12: Effect of tuning the quorum size k of the CCE majority vote.

must be unanimous, as the CCE eventually rejects every example—causing F_1, Precision, and Recall to be undefined for kept elements. As more folds of the CCE are required to agree with each other before a decision is accepted, the CCE will reject more elements. If less folds are required, more elements will be accepted. Similarly, the quality of the rejection lessens: more elements are rejected on which the underlying classifier would not have made a mistake. Tuning the majority vote conditions on the calibration set can help find the sweet spot between the performance of kept elements, and the quality—and volume—of rejections.

We further show how CCE can be tuned to be more or less conservative in its rejections...

6.8.2 *Guidance for Choosing Calibration Constraints*

In Section 6.6.4 we formally describe the threshold calibration as an optimization problem in which one metric of interest is maximized or minimized given constraints on another metric. Throughout our evaluation we focus on maximizing the F_1 of kept elements, while keeping a reasonably low rejection rate. We choose 15% after taking into account the size of our dataset and using guidance from Miller et al. [194] to estimate a reasonable labeling capacity.

Recall that the calibration constraints are with respect to the calibration set which ideally exhibits minimal drift. It is clear from our evaluation that as concept drift becomes more severe during a deployment, constraints such as those on the rejection rate will be surpassed to some degree. This is the desired outcome—so long as the performance on rejected elements remains low (*i.e.*, they would likely be misclassified) we would rather reject drifting examples.

Figure 6.13 presents an alternative to the optimization used in our previous experiments which is more appropriate if the rejection rate must be kept low. By finding thresholds that minimize the rejection rate on the calibration set with F_1-Score no less than 0.8, during deployment the rejection rate stays much lower, consistently staying below 10% even as the drift increases. Similar to how the rejection rate begins close to the calibration constraint and then increases in our previous experiments, in this setting the F_1 begins close to the calibration constraint, and then decreases. The overall effect here is that the ICE is more conservative in its rejections: while the F_1 of kept elements decreases as more incorrect predictions are accepted, the ICE rejects only those predictions that are

...and how to choose alternative calibration settings to prioritize different performance goals.

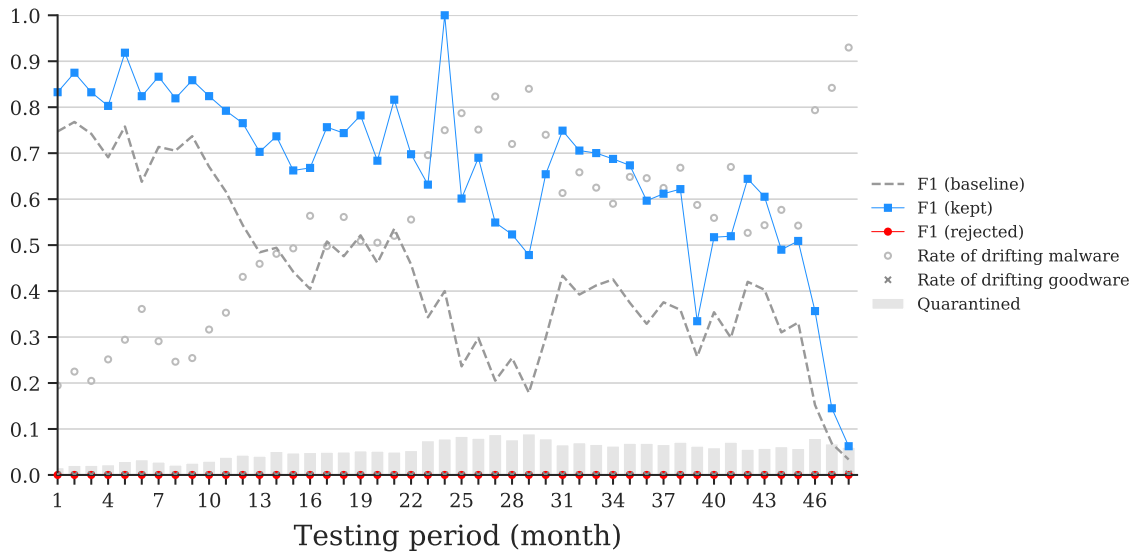


Figure 6.13: F_1-Score of an ICE optimized to find thresholds minimizing the rejection rate with F_1-Score no less than 0.8. This keeps the rejection rate below 10% while sacrificing F_1 performance on kept elements (cf. Figure 6.7 (b)).

most likely to be incorrect, keeping the F_1 of rejected elements at 0.

In summary, to estimate how many rejections will be acceptable, we advise practitioners to consider the expected volume of incoming samples, the available resources for processing quarantined examples, and the lifetime of the classifier before being retrained (as drift will likely increase during this period). Next they should identify which metrics are most important, or non-negotiable, and use these to balance the threshold optimization. As the emergence of concept drift will likely result in the calibration values being surpassed, a ballpark is more important than the exact values.

6.9 Related Work

Conformal evaluation is based on conformal prediction theory, a mechanism for producing predictions that are correct with some guaranteed confidence [253]. Additionally, the ICE and CCE are inspired by inductive [301, 300, 211] and cross-conformal predictors [302], respectively. However, conformal prediction is intended to be used exclusively in settings where the exchangeability assumption holds which makes it unsuitable for adversarial contexts such as malware classification. In this regard, we are the first to 'join the dots' between the conformal prediction of Vovk et al. [301] and the conformal evaluation of Jordaney et al. [139] and show how the violation of conformal prediction's assumptions is detected and exploited by Transcend [139] to detect concept drift.

That work introduced the concept of *conformal evaluation* based on conformal prediction theory and the use of p-values for calibrating and enforcing a rejection strategy for malware classification. However the evaluation artificially simulated concept drift by merging malware datasets which introduced experimental bias [223, 20] (Section 6.7). In our experiments we sample from a

single repository of applications and perform a temporal evaluation to simulate natural concept drift caused by the real evolution of malicious Android apps. Additionally, the role of confidence in thresholding was unclear, and the use of exhaustive grid search to find thresholds was suboptimal compared to our random search. Most significantly, the TCE employed by the original work was not practical for real-world deployments, which we rectify by proposing the ICE and CCE.

Other works have explored alternative solutions to tackling concept drift. As described in Section 6.7.5, Xu et al. [316] propose DroidEvolver, a malware detection system motivated by Transcend [139] that identifies drifting examples based on disagreements between models in an ensemble. As models degrade, the examples identified as drifting are used to update the models in an online fashion.However, we find the drift identification mechanism is inferior to Transcendent and leads to a negative feedback loop, as also evidenced recently by Kan et al. [143]. Other solutions solely adapt to concept drift without using rejection: DroidOL [203] and Casandra [204] use online learning to continually retrain the models, with API call graphs as features. Like all online-trained neural networks, these are susceptible to catastrophic forgetting [105], where performance degrades on older examples as the model attempts to adapt to the new distribution. In Chapter 5 we present a comparison of strategies for combating concept drift, including rejection, incremental retraining, and online learning, illustrating their advantages and disadvantages.

The related task of detecting *adversarial examples* [279, 34, 226] is addressed by Sotgiu et al. [270], who propose a rejection strategy for neural network-based classifiers that identifies anomalies in the latent feature representation of an example at different layers of the neural network. Additionally, Papernot and McDaniel [212] combine a conformal predictor with a k-Nearest Neighbor algorithm to identify low-quality predictions that are indicative of adversarial inputs. However, both of these methods are restricted to deep learning-based image classification.

6.10 Summary

Following the proposal of Transcend [139], rejection strategies have seemed like a promising solution to the issue of concept drift. However, the theoretical soundness, optimal configurations, and real-world practicality were unclear.

In this work we provide a thorough formal treatment of Transcend [139] which acts as the *missing link* between conformal prediction and conformal evaluation. We propose Transcendent, a superset of the original framework which includes novel conformal evaluators that match or surpass the original performance while significantly decreasing the computational cost. We show Transcendent outper-

forms the existing state-of-the-art approaches while generalizing across different malware domains and exploring realistic operational settings.

We envision these improvements will enable researchers and practitioners alike to make use of conformal evaluation to build rejection strategies to improve the quality of security detection pipelines. To accommodate this, we also release our implementation of Transcendent, making Transcend [139] and conformal evaluation available to the security community for the first time.

7 *Conclusions*

This thesis has explored whether machine learning is ready to be used for security detection tasks given the hostile environments that security detectors and classifiers are deployed to. The first part, *Prologue*, gave an overview of the problem, providing a general framework for discussing classification tasks in security settings and outlining fundamental concepts used to describe machine learning in security and the security of machine learning.

The second part, *Adversarial Interactions*, delved deeper into the typical hostile environment itself, and the consequences arising from adversarial activity. We first described the different forms of concept drift, a phenomenon resulting from adversarial activity in which the data distribution diverges from the training data and causes detection performance to deteriorate over time. We discussed the relationship between concept shift and covariate shift to the existence of adversarial examples: objects confidently misclassified as benign by a detector while retaining malicious functionality. We then examined how adversarial examples—traditionally studied in domains such as computer vision—can be realized in more constrained and challenging domains such as malware detection. We demonstrated that realizable evasive Android malware is a realistic and practical threat which also facilitates the development of strong problem-space universal adversarial examples.

The final part, *Detection in a Hostile Environment*, further examined the impact of concept drift in malware detection, but also focused on how to perform detection in spite of its negative effects. Firstly we identified sources of experimental bias which need to be accounted for in order to have a stable evaluation setup that more realistically simulates a dynamic, evolving environment such as that in malware detection. Utilizing this setup, we then proposed an improved framework for identifying and rejecting drifting examples. This framework can be practically used to reduce the burden of misclassifications and help analysts triage detection efforts by quarantining evolving examples.

Given all this, we can conclude that there are numerous limitations to the application of machine learning in security settings. The nature of the security task, in which members of the positive class actively try to evade detection, is alien compared to typical machine

learning settings that have been studied in the past and results in a number of challenges. However despite this, machine learning remains a tantalizing solution to security problems. It's ability to both generalize and scale is still unparalleled and has the potential to vastly reduce the cost of defense, freeing up resources for better end-to-end protections. Reaching its potential, and becoming capable of secure and trustworthy deployment, will require the clever application of new and innovative research to improve robustness and resilience against adversarial behavior.

Core to all future research are *measurement studies* to describe the properties and trends of security data over time [173, 183, 283]. These are essential for better determining the realistic conditions for experiment evaluations—for example the work presented in Chapter 5 would not have been possible without such studies.

The design of *hardened classifiers* [*e.g.*, 77, 133] can be used to improve resilience against specific threats. Orthogonal to these are *robust training methods* [179, 58, 59, 171] which can be extended to produce certifiably robust classifiers—models with a verifiable lower bound on performance versus any attack of a certain strength, making them particularly suitable for safety-critical applications.

Explainability methods can be applied in order to better understand why models make the decisions they do [305, 319, 228, 287, 122]. In this way, explanations can augment the usual numerical decision score by providing descriptions closer to those produced by human analysts, investigators, and reverse engineers. Such methods could also characterize ongoing changes in the feature distribution in ways which are more easily interpretable to a human operator.

As concept drift occurs relative to a particular feature space, one promising future research direction is the development of *robust feature spaces* [325, 288]. Such feature spaces better capture malicious semantics and have fewer redundant dimensions, giving the attacker much less flexibility to fool the classifier by adding superfluous benign attributes or by obscuring their own behavior. As they better describe the core essence of *maliciousness* they are less prone to performance decay induced by changes over time.

We envision that through the invention of more robust methods, and with ingenious engineering, the hostile environment can be tamed to allow for machine learning's benefits to take full effect.

Bibliography

[1] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*, chapter 5. AMLBook, 2012.

[2] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[3] Shruti Agarwal, Hany Farid, Yuming Gu, Mingming He, Koki Nagano, and Hao Li. Protecting world leaders against deep fakes. In *CVPR Workshops*, pages 38–45. Computer Vision Foundation / IEEE, 2019.

[4] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When malware is packin' heat; limits of machine learning classifiers. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles,Techniques, and Tools (2nd Edition)*. Addison Wesley, 2007.

[6] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Are your training datasets yet relevant? - an investigation into the importance of timeline in machine learning-based malware detection. In *Proc. of the International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2015. URL `https://doi.org/10.1007/978-3-319-15618-7_5`.

[7] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 2016.

[8] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proc. of the ACM International Conference on Mining Software Repositories (MSR)*. ACM, 2016.

[9] Sumayah A. Alrwais, Xiaojing Liao, Xianghang Mi, Peng Wang, Xiaofeng Wang, Feng Qian, Raheem A. Beyah, and

Damon McCoy. Under the shadow of sunshine: Understanding and detecting bulletproof hosting on legitimate service provider networks. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[10] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[11] Hyrum S. Anderson and Phil Roth. EMBER: an open dataset for training static PE malware machine learning models. *CoRR*, abs/1804.04637, 2018.

[12] Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static PE machine learning malware models via reinforcement learning. *CoRR*, abs/1801.08917, 2018.

[13] Giuseppina Andresini, Feargus Pendlebury, Fabio Pierazzi, Corrado Loglisci, Annalisa Appice, and Lorenzo Cavallaro. INSOMNIA: Towards concept-drift robustness in network intrusion detection. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2021.

[14] D.A. Andriesse, J.M. Slowinska, and H.J. Bos. Compiler-agnostic function detection in binaries. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[15] Apple. App store stopped more than $1.5 billion in potentially fraudulent transactions in 2020. `https://www.apple.com/newsroom/2021/05/app-store-stopped-over-1-5-billion-in-suspect-transactions-in-2020/`, 2021. Accessed: Sep 2021.

[16] Giovanni Apruzzese and Michele Colajanni. Evading Botnet Detectors Based on Flows and Random Forest with Adversarial Samples. In *Proc. of the IEEE International Symposium on Network Computing and Applications (NCA)*, 2018.

[17] Giovanni Apruzzese, Michele Colajanni, and Mirco Marchetti. Evaluating the effectiveness of Adversarial Attacks against Botnet Detectors. In *Proc. of the IEEE International Symposium on Network Computing and Applications (NCA)*, 2019.

[18] Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George F. Foster, Colin Cherry, Wolfgang Macherey, Zhifeng Chen, and Yonghui Wu. Massively multilingual neural machine translation in the wild: Findings and challenges. *CoRR*, abs/1907.05019, 2019.

[19] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[20] 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 *Proc. of the USENIX Security Symposium*, 2022.

[21] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014.

[22] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. of the International Conference on Machine Learning (ICML)*, volume 80, pages 274–283, 2018.

[23] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 2000.

[24] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2015.

[25] Iain M. Banks. A few notes on the culture. `rec.arts.sf.written`, 1994. Archived at `http://www.vavatch.co.uk/books/banks/cultnote.htm`.

[26] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: learning to recognize functions in binary code. In *Proc. of the USENIX Security Symposium*, 2014.

[27] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification with conformal evaluation. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[28] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015.

[29] Diogo Barradas, Nuno Santos, and Luís E. T. Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *Proc. of the USENIX Security Symposium*, 2018.

[30] Peter L. Bartlett and Marten H. Wegkamp. Classification with a reject option using a hinge loss. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

[31] Richard Bellman. *Adaptive Control Processes - A Guided Tour (Reprint from 1961)*, volume 2045. Princeton University Press, 2015. ISBN 978-1-4008-7466-8. URL `https://doi.org/10.1515/9781400874668`.

[32] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 35(8):1798–1828, 2013.

[33] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research (JMLR)*, 13, 2012.

[34] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 2018.

[35] Battista Biggio, Blaine Nelson, and Pavel Laskov. Support vector machines under adversarial label noise. In *Proc. of Asian Conference on Machine Learning (ACML)*, 2011.

[36] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proc. of the International Conference on Machine Learning (ICML)*, 2012.

[37] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Proc. of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*. Springer, 2013.

[38] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2013.

[39] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[40] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Springer, 2007.

[41] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D.

Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[42] Yazan Boshmaf, Dionysios Logothetis, Georgos Siganos, Jorge Lería, José Lorenzo, Matei Ripeanu, and Konstantin Beznosov. Integro: Leveraging victim prediction for robust fake account detection in osns. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[43] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial Patch. *arXiv preprint arXiv:1712.09665*, 2017.

[44] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard E. Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[45] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, 2011.

[46] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[47] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *Proc. on USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[48] Yulong Cao, Chaowei Xiao, Dawei Yang, Jing Fang, Ruigang Yang, Mingyan Liu, and Bo Li. Adversarial Objects against LiDAR-based Autonomous Driving Systems. *arXiv preprint arXiv:1907.05418*, 2019.

[49] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *Proc. of the Deep Learning and Security Workshop (DLS)*. IEEE, 2018.

[50] Nicholas Carlini and David A. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2017.

[51] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2017.

[52] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, and Aleksander Madry. On evaluating adversarial robustness. *CoRR*, abs/1902.06705, 2019.

[53] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *Proc. of the USENIX Security Symposium*, 2021.

[54] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing PDF parsers in malware detectors. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[55] Tanmoy Chakraborty, Fabio Pierazzi, and V. S. Subrahmanian. EC2: Ensemble Clustering and Classification for Predicting Android Malware Families. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2017.

[56] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority oversampling technique. *Journal of Artificial Intelligence Research (JAIR)*, 2002.

[57] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Special issue on learning from imbalanced data sets. *ACM SIGKDD Explorations Newsletter*, 2004.

[58] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. On training robust PDF malware classifiers. In *Proc. of the USENIX Security Symposium*, 2020.

[59] Yizheng Chen, Shiqi Wang, Weifan Jiang, Asaf Cidon, and Suman Jana. Cost-aware robust tree ensembles for security applications. In *Proc. of the USENIX Security Symposium*, 2021.

[60] Bobby Chesney and Danielle Citron. Deep fakes: A looming challenge for privacy, democracy, and national security. *California Law Review*, 107:1753, 2019.

[61] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 2020.

[62] Clarence Chio and David Freeman. *Machine Learning and Security: Protecting Systems with Data and Algorithms*. O'Reilly Media, Inc., 2018.

[63] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *Proc. of the USENIX Security Symposium*, 2017.

[64] Kenneth T. Co, Luis Muñoz-González, Sixte de Maupeou, and Emil C. Lupu. Procedural noise adversarial examples for black-box attacks on deep convolutional networks. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[65] Igino Corona, Giorgio Giacinto, and Fabio Roli. Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues. *Information Sciences*, 2013.

[66] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3), 1995.

[67] Corinna Cortes, Mehryar Mohri, Michael Riley, and Afshin Rostamizadeh. Sample selection bias correction theory. In *Proc. of the International Conference on Algorithmic Learning Theory (ALT)*, 2008.

[68] Marco Cova, Christopher Krügel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proc. of the International World Wide Web Conference (WWW)*. ACM, 2010.

[69] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research (JMLR)*, 7:551–585, 2006.

[70] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. ZOZZLE: fast and precise in-browser javascript malware detection. In *Proc. of the USENIX Security Symposium*. USENIX Association, 2011. URL `http://static.usenix.org/events/sec11/tech/full_papers/Curtsinger.pdf`.

[71] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *Proc. of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. IEEE, 2013.

[72] Nilesh Dalvi, Pedro Domingos, Sumit Sanghai, Deepak Verma, et al. Adversarial classification. In *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*. ACM, 2004.

[73] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[74] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin J. Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Proc. of the IEEE Workshop on Mobile Security Technologies (MoST)*. IEEE Computer Society, 2016.

[75] Jesse Davis and Mark Goadrich. The relationship between precision-recall and ROC curves. In *Proc. of the International Conference on Machine Learning (ICML)*. ACM, 2006.

[76] Emiliano De Cristofaro. An overview of privacy in machine learning. *arXiv:2005.08679*, 2020.

[77] 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 (TDSC)*, 2017.

[78] 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 *Proc. of the USENIX Security Symposium*, 2019.

[79] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*. Association for Computational Linguistics, 2019.

[80] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[81] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3), 1984.

[82] Jun Du and Charles X Ling. Active learning with human-like noisy oracle. In *Proc. of the IEEE International Conference on Data Mining series (ICDM)*. IEEE, 2010.

[83] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[84] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy*, pages 332–346. IEEE Computer Society, 2012.

[85] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. In *Proc. of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[86] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2018.

[87] William Enck, Machigar Ongtang, and Patrick D. McDaniel. On lightweight mobile phone application certification. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[88] Sebastian Eresheim, Robert Luh, and Sebastian Schrittwieser. The evolution of process hiding techniques in malware-current threats and possible countermeasures. *Journal of Information Processing*, 2017.

[89] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[90] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. Graph embedding based familial analysis of android malware using unsupervised learning. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2019.

[91] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, 2008.

[92] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[93] Tom Fawcett. In vivo spam filtering: A challenge problem for kdd. *ACM SIGKDD Explorations Newsletter*, 2003.

[94] Charles Fefferman, Sanjoy Mitter, and Hariharan Narayanan. Testing the manifold hypothesis. *Journal of the American Mathematical Society*, 29(4):983–1049, 2016.

[95] Steven Feldstein. The global expansion of ai surveillance. Technical report, Carnegie Endowment for International Peace, 2019.

[96] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[97] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[98] Felix Fischer, Huang Xiao, Ching-yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, and Jens Grossklags. Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In *Proc. of the USENIX Security Symposium*, 2019.

[99] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[100] Prahlad Fogla, Monirul I. Sharif, Roberto Perdisci, Oleg M. Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proc. of the USENIX Security Symposium*, 2006.

[101] George Forman. A pitfall and solution in multi-class feature selection for text classification. In *Proc. of the International Conference on Machine Learning (ICML)*, 2004.

[102] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[103] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon M. Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *Proc. of the USENIX Security Symposium*, 2014.

[104] B. Frenay and M. Verleysen. Classification in the presence of label noise: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2014.

[105] Robert M. French and Nick Chater. Using noise to compute error surfaces in connectionist networks: A novel means of reducing catastrophic forgetting. *Neural Computation*, 14(7), 2002. URL `https://doi.org/10.1162/08997660260028700`.

[106] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 2001.

[107] Giorgio Fumera, Ignazio Pillai, and Fabio Roli. Classification with reject option in text categorisation systems. In *Int. Conf. Image Analysis and Processing*. IEEE, 2003.

[108] João Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, 2014.

[109] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2013.

[110] Justin Gilmer, Nicolas Ford, Nicholas Carlini, and Ekin D. Cubuk. Adversarial examples are a natural consequence of test error in noise. In *Proc. of the International Conference on Machine Learning (ICML)*, 2019.

[111] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. MIT press Cambridge, 2016.

[112] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR (Poster)*, 2015.

[113] Google. Permissions on Android. `https://developer.android.com/guide/topics/security/permissions.html`, 2016.

[114] Google. Android Security 2016 Year in Review, Tech. Report, 2017.

[115] Google. `https://www.virustotal.com/`, 2019.

[116] Google. Google play protect: 2.5 billion active devices. `https://www.android.com/intl/en_us/intl/en_uk/play-protect/`, 2020. Accessed: Sep 2020.

[117] Will Grimond and Asheem Singh. A force for good? results from foi requests on artificial intelligence in the police force. Technical report, Royal Society for the encouragement of Arts, Manufactures and Commerce, 2020.

[118] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[119] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017.

[120] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. In *ICLR (Workshop)*, 2015.

[121] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv:1708.06733*, 2017.

[122] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: explaining deep learning based security applications. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[123] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. Borderline-smote: A new over-sampling method in imbalanced data sets learning. In *Advances in Intelligent Computing*, 2005.

[124] David J Hand. Measuring classifier performance: a coherent alternative to the area under the ROC curve. *Machine Learning*, 2009.

[125] Karen Hao. How facebook uses machine learning to detect fake accounts. `https://www.technologyreview.com/2020/03/04/905551/how-facebook-uses-machine-learning-to-detect-fake-accounts`, 2019. Accessed: Sep 2021.

[126] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2009.

[127] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[128] Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: Detection and mitigation of malicious websites with a frozen DOM. In *Proc. of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2011.

[129] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313 (5786):504–507, 2006.

[130] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[131] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2011.

[132] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 125–136, 2019.

[133] Inigo Incer, Michael Theodorides, Sadia Afroz, and David Wagner. Adversarially robust malware detection using monotonic classification. In *Proc. of the ACM International Workshop on Security And Privacy Analytics (IWSPA)*. ACM, 2018.

[134] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. *CoRR*, abs/2008.04480, 2020.

[135] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.

[136] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. of the USENIX Security Symposium*, 2015.

[137] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Jsketch: sketching for java. In *Proc. of the ACM SIGSOFT Conference on Foundations of Software Engineering (FSE)*. ACM, 2015.

[138] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*, 2019.

[139] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proc. of the USENIX Security Symposium*, 2017.

[140] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[141] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan

Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *Found. Trends Mach. Learn.*, 14(1-2):1–210, 2021.

[142] Anil Kamath, Rajeev Motwani, Krishna V. Palem, and Paul G. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1994.

[143] Zeliang Kan, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Investigating labelless drift adaptation for malware detection. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2021.

[144] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *AISec*, pages 99–110. ACM, 2013.

[145] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D. Joseph, and J. Doug Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2015.

[146] Kaspersky Lab. Adwind malware-as-a-service hits more than 400,000 users globally. `https://www.kaspersky.co.uk/blog/adwind-rat/6731/`, 2021. (last visited Jan. 22, 2021).

[147] Auguste Kerckhoffs. La cryptographie militaire. In *Journal des sciences militaires*, 1883.

[148] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K. Robertson, and Engin Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *Proc. of the USENIX Security Symposium*, 2016.

[149] Amin Kharraz, William K. Robertson, and Engin Kirda. Surveylance: Automatically detecting online survey scams. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[150] Marc Khoury and Dylan Hadfield-Menell. On the geometry of adversarial examples. *CoRR*, abs/1811.00525, 2018.

[151] Joohee Kim, Minji Kim, Mi-Sun Lee, Kukjoo Kim, Sangyoon Ji, Yun-Tae Kim, Jihun Park, Kyungmin Na, Kwi-Hyun Bae, Hong Kyun Kim, Franklin Bien, Chang Young Lee, and Jang-Ung Park. Wearable smart sensor systems integrated on soft contact lenses for wireless ocular diagnostics. *Nature Communications*, 8(1), Apr 2017.

[152] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[153] Clemens Kolbitsch, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[154] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *Proc. of the IEEE European Signal Processing Conference (EUSIPCO)*. IEEE, 2018.

[155] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[156] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[157] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 1951.

[158] Bogdan Kulynych, Jamie Hayes, Nikita Samarin, and Carmela Troncoso. Evading classifiers in discrete domains with provable optimality guarantees. *CoRR*, abs/1810.10939, 2018.

[159] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *CoRR*, abs/1611.01236, 2016.

[160] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *ICLR (Workshop)*, 2017.

[161] Raphael Labaca-Castro, Luis Muñoz-González, Feargus Pendlebury, Gabi Dreo Rodosek, Fabio Pierazzi, and Lorenzo Cavallaro. Universal adversarial perturbations for malware. *arXiv preprint arXiv:2102.06747*, 2021.

[162] Sebastian Lapuschkin, Stephan Wäldchen, Alexander Binder, Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Unmasking Clever Hans predictors and assessing what machines really learn. *Nature Communications*, 10(1), 2019.

[163] Tracy Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 11(1), 1992.

[164] Pavel Laskov and Nedim Šrndić. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[165] Lastline, Inc. Malware-as-a-service: The 9-to-5 of organized cybercrime. `https://www.lastline.com/blog/malware-as-a-service-the-9-to-5-of-organized-cybercrime/`, 2021. (last visited Jan. 22, 2021).

[166] Eunjo Lee, Jiyoung Woo, Hyoungshick Kim, Aziz Mohaisen, and Huy Kang Kim. You are a game bot!: Uncovering game bots in mmorpgs via self-similarity in the wild. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016.

[167] Sangho Lee and Jong Kim. Warningbird: Detecting suspicious urls in twitter stream. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[168] Mourad Leslous, Valérie Viet Triem Tong, Jean-François Lalande, and Thomas Genet. Gpfinder: tracking the invisible in android malware. In *Proc. of the International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 1999.

[169] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[170] Li Li, Tegawendé Bissyandé, and Jacques Klein. MoonlightBox: Mining android api histories for uncovering release-time inconsistencies. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018.

[171] Linyi Li, Xiangyu Qi, Tao Xie, and Bo Li. Sok: Certified robustness for deep neural networks. *CoRR*, abs/2009.04131, 2020.

[172] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[173] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris

Ioannidis. AndRadar: Fast discovery of android applications in alternative markets. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Springer, 2014.

[174] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proc. of the IEEE Annual Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, 2015.

[175] Henrik Linusson, Ulf Johansson, Henrik Boström, and Tuve Löfström. Classification with reject option using conformal prediction. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. Springer, 2018.

[176] Zachary C. Lipton, Yu-Xiang Wang, and Alexander J. Smola. Detecting and correcting for label shift with black box predictors. In *Proc. of the International Conference on Machine Learning (ICML)*, 2018.

[177] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Hai Li, and Yiran Chen. Dpatch: An Adversarial Patch Attack on Object Detectors. *arXiv preprint arXiv:1806.02299*, 2018.

[178] Daniel Lowd and Christopher Meek. Good word attacks on statistical spam filters. In *Proc. of the Conference on Email and Anti-Spam (CEAS)*, volume 2005, 2005.

[179] Zhaoyang Lyu, Minghao Guo, Tong Wu, Guodong Xu, Kehuan Zhang, and Dahua Lin. Towards evaluating and training verifiably robust neural networks. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[180] Xin Ma, Samad Ahadian, Song Liu, Jingwen Zhang, Shengnan Liu, Teng Cao, Wenbin Lin, Dong Wu, Natan Roberto de Barros, Mohammad Reza Zare, Sibel Emir Diltemiz, Vadim Jucaud, Yangzhi Zhu, Shiming Zhang, Ethan Banton, Yue Gu, Kewang Nan, Sheng Xu, Mehmet Remzi Dokmeci, and Ali Khademhosseini. Smart contact lenses for biosensing applications. *Advanced Intelligent Systems*, 3(5), 2021.

[181] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[182] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a moving target: Addressing web application concept drift. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[183] Federico Maggi, Alessandro Frossi, Stefano Zanero, Gianluca Stringhini, Brett Stone-Gross, Christopher Kruegel, and Giovanni Vigna. Two years of short urls internet measurement: Security threats and countermeasures. In *Proc. of the International World Wide Web Conference (WWW)*. ACM, 2013.

[184] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A pattern recognition system for malicious pdf files detection. In *Proc. of the International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*. Springer, 2012.

[185] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proc. of the ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*. ACM, 2013.

[186] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards robust detection of adversarial infection vectors: Lessons learned in pdf malware. *CoRR*, abs/1811.00830, 2019.

[187] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. MaMaDroid: Detecting android malware by building markov chains of behavioral models. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2017.

[188] Zane Markel and Michael Bilzor. Building a machine learning classifier for malware detection. In *Anti-malware Testing Research Workshop*. IEEE, 2014.

[189] Paul Mason. *PostCapitalism: A guide to our future*. Allen Lane, 2015.

[190] John McHugh. Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 2000.

[191] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining Black-box Android Malware Detection. *Proc. of the IEEE European Signal Processing Conference (EUSIPCO)*, 2018.

[192] Microsoft. New machine learning model sifts through the good to unearth the bad in evasive malware. `https://bit.ly/3cb0XiF`, 2019. Accessed: Sep 2020.

[193] Brad Miller, Alex Kantchelian, Sadia Afroz, Rekha Bachwani, Edwin Dauber, Ling Huang, Michael Carl Tschantz, Anthony D. Joseph, and J. Doug Tygar. Adversarial active

learning. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2014.

[194] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer integration and performance measurement for malware detection. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Springer, 2016.

[195] Bradley Austin Miller. *Scalable Platform for Malicious Content Detection Integrating Machine Learning and Manual Review*. University of California, Berkeley, 2015.

[196] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[197] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 1992. ISBN 0-262-51063-4. URL `http://dl.acm.org/citation.cfm?id=1867135.1867206`.

[198] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[199] Jose G. Moreno-Torres, Troy Raeder, Rocío Alaíz-Rodríguez, Nitesh V. Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. *Pattern Recognition*, 45(1), 2012.

[200] Stylianos Moschoglou, Athanasios Papaioannou, Christos Sagonas, Jiankang Deng, Irene Kotsia, and Stefanos Zafeiriou. Agedb: The first manually collected, in-the-wild age database. In *CVPR Workshops*, pages 1997–2005. IEEE Computer Society, 2017.

[201] Stylianos Moschoglou, Athanasios Papaioannou, Christos Sagonas, Jiankang Deng, Irene Kotsia, and Stefanos Zafeiriou. Agedb: The first manually collected, in-the-wild age database. In *CVPR Workshops*, pages 1997–2005. IEEE Computer Society, 2017.

[202] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[203] Annamalai Narayanan, Yang Liu, Lihui Chen, and Jinliang Liu. Adaptive and scalable android malware detection

through online learning. In *Proc. of the International Joint Conference on Neural Network (IJCNN)*. IEEE, 2016.

[204] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence (TETCI)*, 1(3), 2017.

[205] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[206] Shirin Nilizadeh, Francois Labreche, Alireza Sedighian, Ali Zand, José M. Fernandez, Christopher Kruegel, Gianluca Stringhini, and Giovanni Vigna. POISED: spotting twitter spam off the beaten paths. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[207] Bruno A Olshausen and David J Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

[208] Timon Van Overveldt, Christopher Kruegel, and Giovanni Vigna. Flashdetect: Actionscript 3 malware detection. In *Proc. of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2012.

[209] Mark Palatucci, Dean Pomerleau, Geoffrey Hinton, and Tom M. Mitchell. Zero-shot learning with semantic output codes. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2009.

[210] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[211] Harris Papadopoulos. Inductive conformal prediction: Theory and application to neural networks. In Paula Fritzsche, editor, *Tools in Artificial Intelligence*, chapter 18. IntechOpen, 2008. URL `https://doi.org/10.5772/6078`.

[212] Nicolas Papernot and Patrick D. McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *CoRR*, abs/1803.04765, 2018. URL `http://arxiv.org/abs/1803.04765`.

[213] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016.

[214] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *CoRR*, abs/1605.07277, 2016.

[215] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 582–597, 2016.

[216] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proc. of the ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*, pages 506–519, 2017.

[217] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. SoK: Security and privacy in machine learning. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, April 2018.

[218] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.

[219] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proc. of the USENIX Security Symposium*, 1998.

[220] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[221] 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 (JMLR)*, 12, 2011.

[222] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. Poster: Enabling fair ml evaluations for security. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[223] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: eliminating experimental bias in malware classification across space and time. In *Proc. of the USENIX Security Symposium*, 2019.

[224] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In

Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.

[225] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. on USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[226] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[227] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[228] Lukas Pirch, Alexander Warnecke, Christian Wressnegger, and Konrad Rieck. Tagvet: Vetting malware tags using explainable machine learning. In *EuroSec@EuroSys*, pages 34–40. ACM, 2021.

[229] Plato. *The Symposium*. Penguin, c. 385–370 BC. Penguin Classics edition published 1999, translated by Christopher Gill.

[230] Joaquin Quionero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D. Lawrence. *Dataset Shift in Machine Learning*. The MIT Press, 2009. ISBN 0262170051.

[231] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. *Proc. of the USENIX Security Symposium*, 2019.

[232] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *AAAI Workshops*, 2018.

[233] Edward Raff, Richard Zak, Gary Lopez Munoz, William Fleming, Hyrum S. Anderson, Bobby Filar, Charles Nicholas, and James Holt. Automatic yara rule generation using biclustering. *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2020.

[234] Babak Rahbarinia, Marco Balduzzi, and Roberto Perdisci. Exploring the long tail of (malicious) software downloads. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017.

[235] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*. ACM, 2016.

[236] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2010.

[237] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Icml*, 2011.

[238] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[239] Martin Roesch. Snort - lightweight intrusion detection for networks. In *In Proc. of the 13th USENIX Conference on System Administration*, 1999.

[240] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *Proc. of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2018.

[241] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2012.

[242] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[243] Ahmed Salem, Michael Backes, and Yang Zhang. Don't trigger me! A triggerless backdoor attack against deep neural networks. *CoRR*, abs/2010.03282, 2020.

[244] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. Dynamic backdoor attacks against machine learning models. *CoRR*, abs/2003.03675, 2020.

[245] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K. Penya, Borja Sanz, Carlos Laorden, and Pablo García Bringas. Idea: Opcode-sequence-based malware detection. In *Proc. of the International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2010.

[246] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Hendrik Clausen, Osman Kiraz, Kamer Ali Yüksel,

Seyit Ahmet Çamtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In *Proc. of the IEEE International Conference on Communications (ICC)*. IEEE, 2009.

[247] Vikash Sehwag, Arjun Nitin Bhagoji, Liwei Song, Chawin Sitawarin, Daniel Cullina, Mung Chiang, and Prateek Mittal. Better the devil you know: An analysis of evasion attacks using out-of-distribution adversarial examples. *CoRR*, abs/1905.01726, 2019.

[248] Vikash Sehwag, Arjun Nitin Bhagoji, Liwei Song, Chawin Sitawarin, Daniel Cullina, Mung Chiang, and Prateek Mittal. Analyzing the robustness of open-world machine learning. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2019.

[249] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2), 1996. DOI: 10.1016/0004-3702(95)00045-3. URL `https://doi.org/10.1016/0004-3702(95)00045-3`.

[250] Burr Settles. Active learning literature survey. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2012.

[251] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. Explanation-guided backdoor poisoning attacks against malware classifiers. In *Proc. of the USENIX Security Symposium*, 2021.

[252] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6106–6116, 2018.

[253] Glenn Shafer and Vladimir Vovk. A tutorial on conformal prediction. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

[254] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *Proc. of the Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2009.

[255] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[256] Yun Shen, Enrico Mariconti, Pierre-Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events

through deep learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[257] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *Proc. of the USENIX Security Symposium*, 2015.

[258] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[259] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[260] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *Proc. of the USENIX Security Symposium*, 2019.

[261] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2015.

[262] Payap Sirinam, Mohsen Imani, Marc Juárez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[263] Michael R. Smith, Nicholas T. Johnson, Joe B. Ingram, Armida J. Carbajal, Ramyaa Ramyaa, Evelyn Domschot, Christopher C. Lamb, Stephen J. Verzi, and W. Philip Kegelmeyer. Mind the gap: On bridging the semantic gap between machine learning and information security. *CoRR*, abs/2005.01800, 2020.

[264] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2012.

[265] Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[266] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing and Management*, 2009.

[267] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2010.

[268] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, and Tadayoshi Kohno. Physical adversarial examples for object detectors. In *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[269] Jonghyuk Song, Sangho Lee, and Jong Kim. Crowdtarget: Target-based detection of crowdturfing in online social networks. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[270] Angelo Sotgiu, Ambra Demontis, Marco Melis, Battista Biggio, Giorgio Fumera, Xiaoyi Feng, and Fabio Roli. Deep neural rejection against adversarial examples. *EURASIP Journal on Information Security*, 2020, 2020. URL `https://doi.org/10.1186/s13635-020-00105-y`.

[271] Nedim Šrndic and Pavel Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2013.

[272] Nedim Srndic and Pavel Laskov. Hidost: a static machine-learning-based detector of malicious files. *EURASIP J. Inf. Secur.*, 2016:22, 2016.

[273] Pierre Stock and Moustapha Cissé. Convnets and imagenet beyond accuracy: Understanding mistakes and uncovering biases. In *Proc. of the European Conference on Computer Vision (ECCV)*, 2018.

[274] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady Paths: Leveraging surfing crowds to detect malicious web pages. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013.

[275] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems With Applications*, 2014.

[276] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. DroidSieve: Fast and accurate classification of obfuscated android malware. In *Proc. of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*, March 2017.

[277] Octavian Suciu, Radu Mărginean, Yiğitcan Kaya, Hal Daumé III, and Tudor Dumitraş. When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks. *Proc. of the USENIX Security Symposium*, 2018.

[278] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[279] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR (Poster)*, 2014.

[280] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[281] Gil Tahan, Lior Rokach, and Yuval Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *Journal of Machine Learning Research (JMLR)*, 2012.

[282] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[283] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 2017.

[284] Kymie M. C. Tan and Roy A. Maxion. "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2002.

[285] Yuqing Tang, Chau Tran, Xian Li, Peng-Jen Chen, Naman Goyal, Vishrav Chaudhary, Jiatao Gu, and Angela Fan. Multilingual translation with extensible multilingual pretraining and finetuning. *CoRR*, abs/2008.00401, 2020.

[286] Ronghua Tian, Lynn Margaret Batten, Md. Rafiqul Islam, and Steven Versteeg. An automated classification system based on the strings of trojan and virus families. In *Proc. of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[287] Alexander Warnecke Konrad Rieck Tom Ganz, Martin Härterich. Explaining graph neural networks for vulnerability

discovery. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2021.

[288] Liang Tong, Bo Li, Chen Hajaj, Chaowei Xiao, Ning Zhang, and Yevgeniy Vorobeychik. Improving robustness of ML classifiers against realizable evasion attacks using conserved features. In *Proc. of the USENIX Security Symposium*. USENIX Association, 2019.

[289] Antonio Torralba and Alexei A Efros. Unbiased look at dataset bias. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2011.

[290] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *Proc. of the USENIX Security Symposium*, 2016.

[291] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. Adversarial: Perceptual ad blocking meets adversarial machine learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[292] James Tu, Mengye Ren, Sivabalan Manivasagam, Ming Liang, Bin Yang, Richard Du, Frank Cheng, and Raquel Urtasun. Physically Realizable Adversarial Examples for LiDAR Object Detection. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[293] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[294] Phani Vadrevu, Babak Rahbarinia, Roberto Perdisci, Kang Li, and Manos Antonakakis. Measuring and detecting malware downloads in live network traffic. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2013.

[295] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *Proc. of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*. IBM Corp., 1999.

[296] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. Benchmarking Crimes: An emerging threat in systems security. *arXiv preprint*, 2018.

[297] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. SoK: Benchmarking Flaws

in Systems Security. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[298] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[299] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proc. of the International Conference on Machine Learning (ICML)*, 2008.

[300] Vladimir Vovk. Conditional validity of inductive conformal predictors. *Journal of Machine Learning Research (JMLR)*, 92 (2-3), 2013.

[301] Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*. Springer-verlag New York Inc., 2010. ISBN 9781441934710.

[302] Vladimir Vovk, Ilia Nouretdinov, Valery Manokhin, and Alexander Gammerman. Cross-conformal predictive distributions. In *Proc. of the PMLR Workshop on Conformal Prediction and its Applications (COPA)*, volume 91. PMLR, 2018.

[303] Nedim Šrndić and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[304] Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal adversarial triggers for attacking and analyzing NLP. In *EMNLP/IJCNLP*, 2019.

[305] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. Evaluating explanation methods for deep learning in security. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[306] Webroot Inc. Malware as a service: As easy as it gets. `https://www.webroot.com/blog/2016/03/31/malware-service-easy-gets/`, 2021. (last visited Jan. 22, 2021).

[307] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[308] Mark Weiser. Program slicing. In *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE Press, 1981. URL `http://dl.acm.org/citation.cfm?id=800078.802557`.

[309] Gary M Weiss and Foster Provost. Learning when training data are costly: The effect of class distribution on tree induction. *Journal of Artificial Intelligence Research (JAIR)*, 2003.

[310] Karl R. Weiss, Taghi M. Khoshgoftaar, and Dingding Wang. A survey of transfer learning. *Journal of Big Data*, 3:9, 2016.

[311] William Weiss and Cherie D'Mello. *Fundamentals of model theory*. Topology Atlas, 2000.

[312] David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 1996.

[313] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell. Understanding data augmentation for classification: When to warp? In *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, 2016.

[314] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, and Jian Lu. Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[315] Qixue Xiao, Yufei Chen, Chao Shen, Yu Chen, and Kang Li. Seeing is not believing: Camouflage attacks on image scaling algorithms. In *Proc. of the USENIX Security Symposium*, 2019.

[316] Ke Xu, Yingjiu Li, Robert H. Deng, Kai Chen, and Jiayun Xu. Droidevolver: Self-evolving android malware detection system. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019.

[317] Teng Xu, Gerard Goossen, Huseyin Kerem Cevahir, Sara Khodeir, Yingyezhe Jin, Frank Li, Shawn Shan, Sagar Patel, David Freeman, and Paul Pearce. Deep entity classification: Abusive account detection for online social networks. In *Proc. of the USENIX Security Symposium*, 2021.

[318] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[319] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. CADE: detecting and explaining concept drift samples for security applications. In *Proc. of the USENIX Security Symposium*, 2021.

[320] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015.

[321] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017.

[322] Kai Yu, Tong Zhang, and Yihong Gong. Nonlinear learning using local coordinate coding. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2009.

[323] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *SIGCOMM Computer Communication Review*. ACM, 2014.

[324] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[325] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2020. URL `https://doi.org/10.1145/3372297.3417291`.

[326] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proc. of the ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*. ACM, 2018.

[327] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *Proc. of the USENIX Security Symposium*, 2020.

[328] Yongchun Zhu, Dongbo Xi, Bowen Song, Fuzhen Zhuang, Shuai Chen, Xi Gu, and Qing He. Modeling users' behavior sequences with hierarchical explainable network for cross-domain fraud detection. In *Proc. of the International World Wide Web Conference (WWW)*, 2020.

[329] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[330] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *CoRR*, abs/1911.02685, 2019.

[331] Giulio Zizzo, Chris Hankin, Sergio Maffeis, and Kevin Jones. Adversarial machine learning beyond the image domain. In *Proc. of the ACM Design Automation Conference (DAC)*, 2019.