Practical Inference-Time Attacks Against Machine-Learning Systems and a Defense Against Them

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Department of Electrical and Computer Engineering

Mahmood Sharif

B.Sc., Computer Science, University of Haifa
M.Sc., Computer Science, University of Haifa

Carnegie Mellon University
Pittsburgh, PA

December, 2019

Keywords: Adversarial Examples, Machine Learning, Security, Attacks, Defenses.

For my family and friends.

Committee Members

Lujo Bauer (Co-chair), Carnegie Mellon University

Nicolas Christin (Co-chair), Carnegie Mellon University

Matt Fredrikson, Carnegie Mellon University

Michael K. Reiter, University of North Carolina at Chapel Hill

Acknowledgements

This thesis would not have been possible without many outstanding people—I would like to express my sincere gratitude to them.

Special thanks goes to my advisors, Lujo Bauer and Nicolas Christin—working with them has made me the researcher I am today. Among many things, under their guidance I learned how to approach research problems systematically and how to turn rough ideas to top tier papers (and even where to put hyphens between words in these papers). If I ever mentor students in the future, I hope I will do so while being as competent, gentle, and supportive as they are.

My gratitude also goes to Matt Fredrikson and Mike Reiter, the other committee members. While I have not had the pleasure to collaborate closely with Matt on a research project, I deeply appreciate his feedback and questions that always triggered deep thoughts. With Mike, I had the privilege to collaborate on several projects, including all the ones that are presented in this thesis. I am thankful for the optimism he showed when research problems turned more difficult than we had initially thought, for his listening skills and useful feedback, and for his eye-catching title proposals.

Over the last five years, I had the fortune to acquire many additional collaborators, including: Akihiro Nakarai, Akira Yamada, Anupam Das, Ayumu Kubota, Billy Melicher, Chris Gates, Daniel Kats, Janos Szurdi, Josh Tan, Jumpei Urakawa, Keane Lucas, Kevin Roundy, Limin Jia, Matteo Dell'Amico, Matthias Beckerle, Michelle Mazurek, Mihai Christodorescu, Pedro Leon, Saurabh Shintre, Sruti Bhagavatula, Yukiko Sawaya, and Zack Weinberg. I am grateful for their help and for the opportunity to learn from them.

My family has been supportive of me throughout the entirety of the PhD journey. My parents gave me the tools and instilled me with the courage to pursue my dreams, my grandparents pushed me to learn and be curious since I can remember myself, my siblings were always helpful (if only by putting a smile on my face), and Bassma Khamaisi, my fiancé, penetrated my life in the middle of the journey and immediately made it more enjoyable in every possible way. I thank them for their support and for making my life more meaningful.

Moving to Pittsburgh to do a PhD gave me the opportunity to develop deep friendships that will remain with me for life. In particular, Aaron Harlap, Alessandro Giordano, Aymeric Fromherz, Janos Szurdi, Josh Tan, Manar Saria, Mohammad Alawiah, Orsi Kovács, Pardis Emami-Naeini, Simon Weiss, and Sruti Bhagavatula were always there for me when I needed them. I thank them for being my second family away from home.

Last, I thank the remaining CyLab faculty for the advice they gave me throughout the years, the CyLab staff for always being ready to help, my fellow CyLab students for being friendly and kind, and my squash and soccer friends for tolerating my limited skills.

The work presented in this thesis was supported in part by the Multidisciplinary University Research Initiative (MURI) Cyber Deception grant; by NSF grant 1801391 and 1801494; by the National Security Agency under Award No. H9823018D0008; by gifts from Google and Nvidia, and from Lockheed Martin and NATO through Carnegie Mellon CyLab; and by a CyLab Presidential Fellowship and a NortonLifeLock Research Group Fellowship.

Abstract

Prior work has shown that machine-learning algorithms are vulnerable to evasion by so-called adversarial examples. Nonetheless, the majority of the work on evasion attacks has mainly explored L_p-bounded perturbations that lead to misclassification. From a computer-security perspective, such attacks have limited practical implications. To fill the gap, we propose evasion attacks that satisfy *multiple objectives*, and show that these attacks pose a practical threat to computer systems. In particular, we demonstrate how to produce adversarial examples against state-of-the-art face-recognition and malware-detection systems that simultaneously satisfy multiple objectives (e.g., smoothness and robustness against changes in imaging conditions) to mislead the systems in practical settings. Against face recognition, we develop a systematic method to automatically generate attacks, which are realized through printing a pair of eyeglass frames. When worn by attackers, the eyeglasses allow them mislead face-recognition algorithms to evade recognition or impersonate other individuals. Against malware detection, we develop an attack that guides binary-diversification tools via optimization to transform binaries in a functionality preserving manner and mislead detection.

The attacks that we initially demonstrate achieve the desired objectives via ad hoc optimizations. We extend these attacks via a general framework to train a generator neural network to emit adversarial examples satisfying desired objectives. We demonstrate the ability of the proposed framework to accommodate a wide range of objectives, including imprecise ones difficult to model, in two application domains. Specifically, we demonstrate how to produce adversarial eyeglass frames to mislead face recognition with better robustness, inconspicuousness, and scalability than previous approaches, as well as a new attack to fool a handwritten-digit classifier.

Finally, to protect computer-systems from adversarial examples, we propose n-ML—a novel defense that is inspired by n-version programming. n-ML trains an ensemble of n classifiers, and classifies inputs by a vote. Unlike prior approaches, however, the classifiers are trained to classify adversarial examples differently than each other, rendering it very difficult for an adversarial example to obtain enough votes to be misclassified. In several application domains (including face and street-sign recognition), we show that n-ML roughly retains the benign classification accuracies of state-of-the-art models, while simultaneously defending against adversarial examples (produced by our framework, or L_p-based attacks) with better resilience than the best defenses known to date and, in most cases, with lower inference-time overhead.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite their high performance in various challenging tasks (e.g., playing Go [195] and poker [24], recognizing thousands of people in images [159], and diagnosing skin cancer [56]), machine-learning (ML) algorithms in general, and deep neural networks (DNNs) in particular, are vulnerable to evasion attacks at inference time [18, 82, 124, 206]: attackers can mislead ML algorithms by strategically manipulating inputs. Such adversarially crafted inputs that mislead ML algorithms are often called adversarial examples. Fig. 1.1 depicts an adversarial example against an object-recognition DNN.

Prior studies of evasion attacks focused primarily on the so-called *imperceptible* attacks, in settings where the objective is to find perturbations with small L_p-norms[1] and the attackers have complete control over the inputs (e.g., [18, 66, 206]). While keeping the perturbations' L_p-norms small aims to ensure the attacks' imperceptibility to humans, it is imperfect at doing so, as our work as well as others' have shown [185, 189]. In particular, perturbations with small L_p-norms may produce inputs that would be erroneously considered as imperceptible [185, 189]. Additionally, L_p-bounded perturbations cannot produce various kinds of adversarial examples (e.g., by rotating images) [55, 78, 90, 174, 222]. Said differently, ensuring that the differences between purported adversarial examples and their benign counterparts have small L_p-norms is neither necessary nor sufficient for producing successful attacks [189]. Moreover, and perhaps more importantly, prior attacks are often impractical. For example, an attacker attempting to fool face recognition may only control her physical appearance, and cannot precisely control the exact value of any pixel in the face image being classified.

This thesis aims to fill the gap by proposing evasion attacks with *multiple objectives* (specifically, ones other than minimizing the L_p-norms of perturbations) that pose a practical threat to state-of-the-art ML systems, including ones for face recognition and malware detection. Then, to mitigate the risks of adversarial examples and improve the security of ML systems at inference time, the thesis presents a novel defense that uses diversified models in an ensemble to detect adversarial examples.

[1] Formally, the L_p-norm of a perturbation vector r (the difference between the perturbed and the benign input) is denoted by $||r||_p$, and is defined as $||r||_p = (\sum_i |r_i^p|)^{\frac{1}{p}}$. Typically in the adversarial ML literature, L_0 (the number of non-zero entries in r) [31, 157], L_2 (the Euclidean norm of r) [31, 206], and L_∞ (the largest absolute values of r's entries) [31, 66] are used to quantify the imperceptibility of evasion attacks.

(a) (b) (c)

Figure 1.1: An illustration of an adversarial example created using Szegedy et al.'s method [206]. The adversarial example targets a DNN proposed by Chatfield et al. [33]. (a) A benign image of a lion that is correctly classified (probability assigned to the correct class, p=0.99). (b) An adversarially perturbed image that is misclassified as a pelican (p=0.97). (c) Absolute value of the pixel-wise differences between the adversarial and the benign image. All differences are close to zero (i.e., almost black).

1.1 Thesis Statement

Adversarial examples pose a practical security threat to ML systems at inference time. Such adversarial examples can be generated by a general framework, even when the characteristics necessary for their success elude precise specification. Ensembles of diversified models can help mitigate attacks better than the state of the art.

To support this statement, we present four research studies:

Physical-world attacks against face recognition[2] This study demonstrates how to produce eyeglass frames that, when printed and worn, can be used by attackers to *dodge* recognition (i.e., get arbitrarily misclassified as other individuals) or *impersonate* specific individuals. To make the eyeglass frames physically realizable we require that they satisfy multiple objectives: that they have smooth transitions between neighboring pixels (similarly to real images); that their colors can be printed by a commodity printer; and that they would lead to misclassification under volatile imaging conditions (e.g., different poses). We develop a systematic method to produce adversarial eyeglass frames, and demonstrate successful attacks against state-of-the-art DNNs for face recognition of the VGG architecture [159] (e.g., $\geq$80% success rate in all dodging attempts in the physical environment).

Functionality-preserving attacks against malware detection[3] This work demonstrates how attackers can alter binaries to fool state-of-the-art DNNs for malware detection from binaries' raw bytes [107, 165]. In this domain, attackers face a non-trivial objective: in addition to misleading the malware detectors, any alteration of a binary must not harm its original, intended, functionality. We show that this objective can be accommodated by the process of producing adversarial examples. Specifically, we propose an attack that guides binary-diversification tools via optimization to mislead

[2]This work was published in the proceedings of the ACM Conference on Computer and Communications Security (CCS), 2016 [191].

[3]This work is currently under submission [193].

DNNs for malware detection while preserving the functionality of binaries. Unlike other attacks on such DNNs (e.g., [101, 108]), ours manipulates instructions that are a functional part of the binary, which makes it particularly challenging to defend against. We evaluated our attack against three DNNs in different settings, and found that it can often achieve success rates near 100%.

A general framework for attacks with objectives[4] The first two attacks model the various objectives they consider in an ad hoc fashion. We complement them by proposing a general framework for capturing objectives in the process of generating adversarial examples. The framework builds on recent work in generative adversarial networks (GANs) [65] to train an attack *generator*—a neural network that can generate attack instances that meet certain objectives. The framework is not only general, but, unlike previous attacks, produces a large number of diverse adversarial examples that meet the desired objectives. We refer to the framework using the anagram AGNs, for *adversarial generative nets*, due to its basis in GANs. Via experiments in two application domains—face and digit recognition—we demonstrate AGNs' flexibility at incorporating various objectives in the process of producing adversarial examples, and particularly objectives that elude precise specification, but can be demonstrated via examples (e.g., inconspicuousness). Compared to the previous attack on face recognition, AGNs are able to produce adversarial eyeglasses that are more inconspicuous and robust to changes in the imaging environment.

Mitigating adversarial examples via ensembles of topologically manipulated classifiers[5] The last part of the thesis presents n-ML, a defense against adversarial examples that is inspired by n-version programming [13, 34, 46]. While n-version programming relies on independent programs to detect unexpected inputs that trigger bugs or exploit vulnerabilities, n-ML uses an ensemble of diversified DNNs to detect adversarial examples. At inference time, n-ML classifies inputs by a vote of the DNNs, where each is trained via the *topology manipulation* method—a new method we propose to train DNNs to classify benign inputs correctly and to (mis)classify adversarial examples differently than one another, according to a certain specification. Consequently, the DNNs mostly agree when classifying benign inputs, and disagree on adversarial examples, thus creating an opportunity to detect the latter. We evaluated n-ML in various application domains (including face and street-sign recognition) and found that: *1)* it roughly retains the benign accuracies of state-of-the-art models; *2)* it outperforms prior defenses (e.g., [126, 127]) at defending against adversarial examples (produced via traditional L_p-based attacks or AGNs); and *3)* it has lower inference-time overhead than the majority of prior defenses we tested.

1.2 Outline

Before delving into the technical content of the thesis, we cover the relevant background and related work (Chap. 2). Subsequently, we describe our attacks on face recognition (Chap. 3) and malware detection (Chap. 4). Next, we present our AGNs framework

[4] This work was published in the ACM Transactions on Privacy and Security (TOPS), 2019 [192].
[5] This work is currently under submission [190].

(Chap. 5), followed by the n-ML defense (Chap. 6). We close the thesis with a summary of our contributions and a discussion of future work (Chap. 7).

Chapter 2

Background and Related Work

In this chapter, we provide background and describe prior work on inference-time attacks on ML algorithms and defenses against them. We first present typical attacks that minimize L_p-norms of perturbations, followed by a discussion of attacks with different objectives in the image and malware classification domains. Subsequently, we provide an overview of prior defenses and link them to our work. Finally, we wrap the chapter with a discussion of threat models that are typically considered in the area.

2.1 Attacks Minimizing L_p-norms

In concurrent research efforts, Szegedy et al. and Biggio et al. showed how to systematically find adversarial examples to fool DNNs [18, 206]. Given an input x that is classified to $\mathbb{F}(x)$ by the DNN, Szegedy et al.'s goal was to find a perturbation r of minimal L_2-norm such that $x + r$ would be classified to a desired target class c_t. They showed that, when the DNN function, $\mathbb{F}(\cdot)$, and the norm function are differentiable, finding the perturbation can be formalized as an optimization problem that can be solved by standard techniques [150]. Differently from the minimal perturbations proposed by Szegedy et al., Biggio et al. focused on finding perturbations that would significantly increase the ML algorithm's confidence in the target class [18]. Attacks akin to Biggio et al.'s are more suitable for the security domain, where one may be interested in assessing the security of algorithms or systems under worst-case attacks [19, 63].

More efficient algorithms were later proposed for finding even more imperceptible adversarial examples (using different L_p-norms), or ones with higher confidence [31, 66, 83, 142, 145, 157, 176]. For example, Carlini and Wagner experimented with different formulations of the optimization's objective function for finding adversarial examples [31]. They proposed solving the following optimization problem to find adversarial examples with perturbations of small L_2-norms that target class c_t:

$$\arg\min_{r} \; Loss_{cw}^{targ}(x + r, c_t) + \kappa \cdot ||r||_2$$

where x is a benign input, r is the perturbation, and κ helps tune the L_2-norm of the perturbation. $Loss_{cw}^{targ}$ is roughly defined as:

$$Loss_{cw}^{targ}(x + r, c_t) = \max_{c \neq c_t}\{\mathbb{L}_c(x + r)\} - \mathbb{L}_{c_t}(x + r)$$

where $\mathbb{L}_c$ is the output for class c at the logits of the DNN—the output of the one-before-last layer. Minimizing $Loss_{cw}^{targ}$ leads $x + r$ to be (mis)classified as c_t. As this formulation targets a specific class c_t, the resulting attack is commonly referred to as a *targeted attack*. In the case of evasion where the aim is to produce an adversarial example that is misclassified as any class but the true class (commonly referred to as *untargeted attack*), $Loss_{cw}^{untarg}$ is used:

$$Loss_{cw}^{untarg}(x + r, c_x) = \mathbb{L}_{c_x}(x + r) - \max_{c \neq c_x}\{\mathbb{L}_c(x + r)\}$$

where c_x is the true class of x. We use $Loss_{cw}$ as a loss function to mislead DNNs in several attacks (e.g., in Chap. 4).

The Projected Gradient Descent (*PGD*) attack is considered as the strongest first-order attack (i.e., an attack that uses gradient decent to find adversarial examples) [129]. Starting from any random point close to a benign input, *PGD* consistently finds perturbations with constrained L_2- or L_∞-norms that achieve roughly the same loss value. Given a benign input x, a loss function, say $Loss_{cw}$, a target class c_t, a step size α, and an upper bound on the norm ϵ, *PGD* iteratively updates the adversarial example until a maximum number of iterations is reached such that:

$$x_{i+1} = Project_{x,\epsilon}(x_i + \alpha sign(\nabla_x Loss_{cw}(x_i, c_t)))$$

where $Project_{x,\epsilon}(\cdot)$ projects vectors on the ϵ-ball around x (e.g., by clipping for L_∞-norm), and $\nabla_x Loss_{cw}(\cdot)$ denotes the gradient of the loss function. *PGD* starts from a random point x_0 in the ϵ-ball around x, and stops after a fixed number of iterations (20–100 iterations are typical [129, 188]). We compare our AGNs framework with a generalized variant of the *PGD* attack in Chap. 5, and use *PGD* to evaluate our n-ML defense in Chap. 6.

Close to our AGNs framework (Chap. 5), Baluja and Fischer [15] proposed to train an auto-encoding neural network that takes an image as input and outputs a perturbed version of the same image that would be misclassified. Follow-up research efforts concurrent to ours proposed to train generative neural networks to create adversarially perturbed images that lead to misclassification [161, 221, 234]. Differently than our work, these attacks require only that the adversarial perturbations have small L_p-norms.

2.2 Attacks with Other Objectives

While the majority of the work on evasion attacks against ML algorithms mainly considers perturbations with bounded L_p-norms, similarity as measured by L_p distances does not fit human perceptions [185, 189]. Moreover, attacks with bounded L_p-norms are not suitable for evading ML systems in practical settings. In response, research efforts explored attacks with other objectives. We describe such efforts in the image- and malware-classification domains in what follows.

2.2.1 Image Domain

Imperceptibility via Non-L_p Metrics To ensure imperceptibility, researchers proposed producing adversarial examples that are similar to their benign counterparts as measured by metrics other than L_p-norms (e.g., [55, 78, 90, 174, 220, 222]). For instance, Rozsa et al. proposed to produce adversarial examples that have high *structural similarity* (a standard metric for measuring similarity between images) with their benign counterparts [174]. Akin to L_p-norms, other metrics do not fit human perception [185, 189].

Universality Moosavi et al. showed how to find universal adversarial perturbations, which lead not just one image to be misclassified, but a large set of images [144]. Universal perturbations improve our understanding of DNNs' limitations, as they show that adversarial examples often lie in fixed directions (in the image RGB space) with respect to their corresponding benign inputs. Differently from that work, in Chap. 5 we explore universal attacks that are both inconspicuous and constrained to a small region.

Physical Realizability Kurakin et al. demonstrated that imperceptible adversarial examples can fool DNNs even if the input to the DNN is an image of the adversarial example printed on paper [112]. Their attack may not be practical in a security-sensitive setting.

Following our work on physical-world attacks against face recognition (Chap. 3), researchers showed that adversarial examples can be physically realized to fool other image-recognition classifiers (e.g., [11, 36, 57, 197, 236]). For example, Evtimov et al. showed that specially crafted patterns printed and affixed to street signs can mislead DNNs for street-sign recognition [57], while Sitawarin et al. showed how to create seemingly innocuous logos and ads that would be (mis)interpreted as street signs by street-sign recognition [197].

Another line of work attempts to achieve privacy from face-recognition systems by completely avoiding *face detection* [73, 228]. Essentially, face detection finds subwindows in images that contain faces, which are later sent for processing by face-recognition systems. Consequently, by evading detection, one could avoid the post processing of her face image by recognition systems. Differently from our work (Chap. 3 and Chap. 5), the proposed techniques are not inconspicuous: they either use excessive makeup [73] or attempt to blind the camera using light-emitting eyeglasses [228].

The susceptibility to physical-world attacks is not restricted to learning systems that operate on visual inputs [27, 29, 42, 53, 162, 182, 233]. For instance, researchers showed that speech-recognition systems can be misled to interpret sounds unintelligible to humans as actual commands [29].

2.2.2 Malware Domain

Preserving Functionality In the realm of malware detection using ML classifiers, researchers proposed attacks to evade ML-based malware classifiers while preserving the (malicious) functionality of the malware. Some attacks (e.g., [48, 200, 215, 227]) tweak malware to mimic benign files (e.g., by adding benign code-snippets to malicious PDF

files). Other attacks (e.g., [4, 49, 68, 79, 101, 108, 205]) tweak malware using gradient-based optimizations or generative methods (e.g., to find which APIs to import). A third type of attacks uses a combination of mimicry and gradient-based optimizations [172].

Unlike some of the prior work (e.g., [4, 172, 215]) which studied attacks against dynamic ML-based malware detectors, in Chap. 4 we explore attacks that target DNNs for malware detection from raw bytes (i.e., static detection methods) [107, 165]. Furthermore, the attacks we explore do not take advantage of weaknesses in the feature-extraction process by introducing adversarially crafted bytes to unreachable regions of the binaries [101, 108, 205] (which may be possible to detect and sanitize statically, see Chap. 4), or by mangling bytes in the headers of binaries [49] (which can be stripped before classification [171]). Instead, the attacks we propose transform the original code of binaries in a functionality-preserving manner to achieve misclassification.

More traditionally, attackers use various obfuscation techniques to evade malware detection. Packing [20, 173, 207, 210]—encrypting binaries' code and data, and then decrypting them at run time—is commonly used to hide malicious content from static detection methods. We only consider unpacked binaries, as is often the case for static analysis methods [20, 107]. Attackers also obfuscate binaries by substituting instructions with others or altering the control flow graphs of binaries [38, 39, 89, 207]. We found that such obfuscation methods do not fool the malware-detection DNNs when applied naïvely. To address this, our attacks leverage stochastic optimization techniques to guide the transformation of binaries and mislead malware detection.

Perhaps most closely related to our work on misleading malware detection is the recent work on misleading ML algorithms for authorship attribution [138, 163]. Meng et al. proposed an attack to mislead authorship attribution at the binary level [138]. Unlike the attacks we propose, Meng et al. leverage weaknesses in feature extraction and modify debug information and non-loadable sections to fool the ML models. Furthermore, their method leaves a conspicuous footprint that the binary was modified (e.g., by introducing multiple data and code sections to the binaries). While this is potentially acceptable for evading author identification, it may raise suspicion when evading malware detection. Quiring et al. recently proposed an attack to mislead authorship attribution from source code [163]. In a similar spirit to our work, their attack leverages an optimization algorithm to guide code transformations that change syntactic and lexical features of the code (e.g., switching between `printf` and `cout`) to mislead ML algorithms for authorship attribution.

2.3 Defending ML Algorithms

Researchers proposed a variety of defenses to mitigate ML algorithms' vulnerability to adversarial examples. Roughly speaking, defenses can be categorized as: adversarial training, certified defenses, attack detection, or input transformation. In addition, similarly to n-ML, some defenses leverage randomness or model ensembles as defesive mechanisms. In what follows, we provide an overview of the different categories and mechanisms.

Adversarial Training Augmenting the training data with correctly labeled adversarial examples that are generated throughout the training process increases models'

robustness to attacks [66, 91, 92, 113, 129, 188, 206]. The resulting training process is commonly referred to as *adversarial training*. In particular, adversarial training with *PGD* (Adv_{PGD}) [129, 188] is one of the most effective defenses to date—we compare it with our proposed n-ML defense (Chap. 6).

Certified Defenses Some defenses attempt to certify the robustness of trained ML models (i.e., provide provable bounds on models' errors for different perturbation magnitudes). Certain certified defenses estimate how DNNs transform L_p balls around benign examples via convex shapes, and attempt to force classification boundaries to not cross the shapes (e.g., [103, 141, 166]). These defenses are less effective than adversarial training with *PGD* [179]. Other defenses estimate the output of the so-called *smoothed classifier* by classifying many variants of the input after adding noise at the input or intermediate layers [43, 117, 122, 178]. The resulting smoothed classifier, in turn, is proven to be robust against perturbations of different L_2-norms. Unfortunately, such defenses do not provide guarantees against other attacks (e.g., physically realizable attacks, or ones with L_∞-bounded adversarial perturbations), and perform less well against them in practice [43, 117].

Attack Detection Similarly to n-ML, there were past proposals for defenses to detect the presence of attacks [2, 60, 67, 80, 125, 126, 137, 139, 152, 217]. While adaptive attacks have been shown to circumvent some of these defenses [10, 30, 125], detectors often significantly increase the magnitude of the perturbations necessary to evade DNNs and detectors combined [30, 126]. In Chap. 5, we show that AGNs can fool face-recognition DNNs that are defended by Metzen et al.'s detector [139] (the leading detector at the time of the work), and, in Chap. 6, we compare n-ML with detection methods based on Local Intrinsic Dimensionality (*LID*) [127] and Network Invariant Checking (*NIC*) [126], which are currently the leading methods for detecting adversarial examples.

Metzen et al. proposed to train a neural network to detect adversarial examples [139]. The detector would take its input from an intermediate layer of a DNN and decide whether the input is adversarial. Carlini and Wagner showed that an adaptive attack, different than the ones on which the detector was originally evaluated against, can evade detection [30]. The attack we show in Chap. 5 follows similar principles.

The *LID* detector uses a logistic regression classifier to tell benign and adversarial inputs apart. The input to the classifier is a vector of *LID* statistics that are estimated for every intermediate representation computed by the DNN. This approach is effective because the *LID* statistics of adversarial examples are presumably distributed differently than those of benign inputs [127].

The *NIC* detector expects certain invariants in DNNs to hold for benign inputs. For example, it expects the provenance of activations for benign inputs to follow a certain distribution. To model these invariants, a linear logistic regression classifier performing the same task as the original DNN is trained using the representation of every intermediate layer. Then, for every pair of neighboring layers, a one-class Support Vector Machine (*oc-SVM*) is trained to model the distribution of the output of the layers' classifiers on benign inputs. Namely, every *oc-SVM* receives concatenated vectors of probability estimates and emits a score indicative of how similar the vectors are to the benign distribution. The scores of all the *oc-SVM*s are eventually combined

to derive an estimate for whether the input is adversarial. In this manner, if the output of two neighboring classifiers on an image, say that of a bird, is (*bat*, *bird*), the input is likely to be benign (as the two classes are similar and likely have been observed for benign inputs during training). However, if the output is (*car*, *bird*), then it is likely that the input is adversarial.

Input Transformation Certain defenses suggest to transform inputs (e.g., via quantization) to sanitize adversarial perturbations before classification [70, 121, 137, 180, 199, 223, 226]. The transformations often aim to hinder the process of computing gradients for the purpose of attacks. In practice, however, it has been shown that attackers can adapt to circumvent such defenses [9, 10, 75].

Randomness Defenses often leverage randomness to mitigate adversarial examples. As previously mentioned, some defenses inject noise at inference time at the input or intermediate layers [43, 80, 117, 122, 178]. Differently, Wang et al. train a hierarchy of layers, each containing multiple paths, and randomly switch between the chosen paths at inference time [219]. Other defenses randomly drop out neurons, shuffle them, or change their weights, also while making inferences [60, 217]. Unlike all these, n-ML uses randomness at *training time* to control how a set of DNNs classify adversarial examples at inference time and uses the DNNs strategically in an ensemble to deter adversarial examples.

Ensembles Similarly to n-ML, several prior defenses suggested using ensembles to defend against adversarial examples. Abbasi and Gagné proposed to measure the disagreement between DNNs and specialized classifiers (i.e., ones that classify one class versus all others) to detect adversarial examples [2]. An adaptive attack to evade the specialized classifiers and the DNNs simultaneously can circumvent this defense [75]. Vijaykeerthy et al. train DNNs sequentially to be robust against an increasing set of attacks [212]. However, they only use the final model at inference time, while we use an ensemble containing several models for inference. A meta-defense by Sengupta et al. strategically selects a model from a pool of candidates at inference time to increase benign accuracy while deterring attacks [186]. This defense is effective against black-box attacks only (see Sec. 2.4). Other ensemble defenses propose novel training or inference mechanisms, but do not achieve competitive performance [93, 153, 204].

Recent papers [128, 225, 232] proposed defenses that, like n-ML, are motivated by n-version programming [13, 34, 46]. In a nutshell, n-version programming aims to provide resilience to bugs and attacks by running n ($\geq$2) variants of independently developed, or diversified, programs. These programs are expected to behave identically for normal (benign) inputs, and differently for unexpected inputs that trigger bugs or exploit vulnerabilities. When one or more programs behaves differently than the others, an unexpected (potentially malicious) input is detected. In the context of ML, defenses that are inspired by n-version programming use ensembles of models that are developed by independent parties [232], different inference algorithms or DNN architectures [128, 225], or models that are trained using different training sets [225]. In all cases, the models are trained via standard training techniques. Consequently, the defenses are often vulnerable to attacks. Moreover, prior work is limited to specific applications (e.g., speech recognition [232]). In contrast, we train the models comprising the n-ML

ensembles via a novel training technique, and our work is conceptually applicable to any domain (see Chap. 6).

2.4 Threat Models

Attacks and defenses are typically evaluated in three settings: black-, grey-, and white-box (e.g., [137, 155–157]). The white-box setting is the most common in the literature. In this setting, it is assumed that the attacker has complete knowledge of the deployed ML model and the defense used to protect it, if any, including their parameters. In consequence, the attacker can leverage her knowledge to perform the strongest evasion attack against the model (e.g., by precisely computing gradients to solve an optimization an produce successful adversarial examples). We consider the white-box setting throughout Chaps. 3–6 of this work. In the black-box setting, the attacker only has (possibly limited) query access to the deployed model, and does not know about the existence of a defense, if any is deployed. In this setting, attackers typically follow one of two directions: either they iteratively query the model and gradually refine the adversarial example until successful evasion is accomplished [23, 85] (we consider such attacks in Chaps. 3–4), or they transfer the attack from a surrogate model to which they have complete access to the targeted model [155, 156] (we consider such attacks in Chaps. 4–6). The grey-box setting is a middle ground between the black- and white-box settings. In this setting, the attacker knows when a defense is deployed, and may know the general architecture of the model and the defense. However, the attacker does not know the exact parameters of the model and the defense. In response, the attacker may be able to build a surrogate that better resembles the targeted model and defense to transfer attacks from. We consider the grey-box setting when evaluating the n-ML defense in Chap. 6.

Chapter 3

Physical-World Attacks Against Face Recognition

3.1 Introduction

While the broad use of ML has also caused a rising interest to understand the extent to which ML algorithms and applications are vulnerable to attacks, the early work did not focus on practical attacks (see Chap. 2). For example, as in the spam-detection domain where the attacker can precisely control the contents of emails and the address of the sender, the domains that received the majority of the attention have the characteristic that the adversary is able to precisely control the digital representation of the input to the ML tools. In this chapter, we explore attacks to mislead facial biometric systems at inference time. These systems are widely used for various sensitive purposes, including surveillance and access control [143, 147, 149]. Thus, attackers who mislead them can cause severe ramifications.

In contrast to domains previously explored, attackers who aim to mislead facial biometric systems often do not have precise control over the systems' input. Rather, attackers may be able to control only their (physical) appearance. The process of converting the physical scene into a digital input is not under the attackers' control, and is additionally affected by factors such as lighting conditions, pose, and distance. As a result, it is more difficult for attackers to manipulate or craft inputs that would cause misclassification than it would be, for example, in the domain of spam detection.

Another difficulty that attackers face in the case of facial biometric systems is that manipulating inputs to evade the ML classifiers might be easily observed from outside the systems. For example, attackers can wear an excessive amount of makeup in order to evade a surveillance system deployed at banks [73]. However, these attackers may draw an increased attention from bystanders, and can be deterred by traditional means, e.g., the police.

In the light of these two challenges, we define and study a new class of attacks: attacks that are *physically realizable* and at the same time are *inconspicuous*. In such attacks, the attacker manipulates the physical state that an ML algorithm is analyzing rather than the digitized representation of this state. At the same time, the manipulations of physical state needed by the attacks are sufficiently subtle that they are either

imperceptible to humans or, if perceptible, seem natural and not representative of an attack.

Inconspicuousness We focus, unlike most related work (e.g., [73]), on attacks that are *inconspicuous*, i.e., a person who is physically present at a scene, or a person who looks at the input gathered by a sensor (e.g., by watching a camera feed), should not notice that an attack is taking place.

We believe that this focus on attacks that are not readily apparent to humans is important for two reasons: First, such attacks can be particularly pernicious, since they will be resistant to at least cursory investigation. Hence, they are a particularly important type of attacks to investigate and learn how to defend against. Second, such attacks help the perpetrators (whether malicious or benign) achieve *plausible deniability*; e.g., a person seeking merely to protect her privacy against aggressive use of face recognition by merchants will plausibly be able to claim that any failure of an ML algorithm to recognize her is due to error or chance rather than deliberate subterfuge.

Physical Realizability In this work, we are interested in attacks that can be *physically implemented* for the purpose of fooling facial biometric systems. In addition, we focus on attacks on state-of-the-art algorithms. Previous work on misleading ML algorithms often targets algorithms that are not as sophisticated or as well trained as the ones used in practice (e.g., [61]), leaving the real-world implications of possible attacks unclear. In contrast, we focus on those attacks that pose a realistic and practical threat to systems that already are or can easily be deployed.

We divide the attacks we study into two categories, which differ both in the specific motivations of potential attackers and in the technical approaches for implementing the attacks. The two categories of attacks we are interested in are *dodging and impersonation*. Both dodging and impersonation target face-recognition systems that perform multi-class classification—in particular, they attempt to find the person to whom a given face image belongs.

Impersonation In an impersonation (i.e., targeted) attack, the adversary seeks to have a face recognized as *a specific other face*. For example, an adversary may try to (inconspicuously) disguise her face to be recognized as an authorized user of a laptop or phone that authenticates users via face recognition. Or, an attacker could attempt to confuse law enforcement by simultaneously tricking multiple geographically distant surveillance systems into "detecting" her presence in different locations.

Dodging In a dodging (i.e., untargeted) attack, the attacker seeks to have her face misidentified as *any* other arbitrary face. From a technical standpoint, this category of attack is interesting because causing an ML system to arbitrarily misidentify a face should be easier to accomplish with minimal modifications to the face, in comparison to misclassifying a face as a particular impersonation target. In addition to malicious purposes, dodging attacks could be used by benign individuals to protect their privacy against excessive surveillance.

In this chapter we demonstrate inconspicuous and physically realizable dodging and impersonation attacks against face-recognition systems. We detail the design of eyeglass frames that, when printed and worn, permitted three subjects (specifically, the

author of the thesis and two collaborators) to succeed at least 80% of the time when attempting dodging against state-of-the-art face-recognition systems models. Other versions of eyeglass frames allowed subjects to impersonate randomly chosen targets. For example, they allowed one subject, a white male, to impersonate Milla Jovovich, a white female, 87.87% of the time; a South-Asian female to impersonate a Middle-Eastern male 88% of the time; and a Middle-Eastern male to impersonate Clive Owen, a white male, 16.13% of the time. We also show various other results as extensions to our core methods, in accomplishing impersonation with less information (i.e., in a black-box setting) and in evading face *detection*.

Our contributions in this chapter are threefold:

1. We show how an attacker that knows the internals of a state-of-the-art face-recognition system (i.e., in a white-box setting) can physically realize impersonation and dodging attacks. We further show how these attacks can be constrained to increase their inconspicuousness (Sec. 3.2–3.3).

2. Using a commercial face-recognition system [135], we demonstrate how an attacker that is unaware of the system's internals (i.e., in a black-box setting) is able to achieve inconspicuous impersonation (Sec. 3.4).

3. We show how an attacker can be invisible to facial biometric systems by avoiding detection through the Viola-Jones face detector, the most popular face-detection algorithm [213] (Sec. 3.5).

In Sec. 3.2 we describe our method for generating physically realizable white-box attacks, and detail experiments that show its effectiveness in Sec. 3.3. We demonstrate that similar attacks can be carried out against black-box face-recognition systems in Sec. 3.4, and that they can additionally be used to evade face *detection* in Sec. 3.5. We discuss the implications and limitations of our approach in Sec. 3.6 and conclude with Sec. 3.7.

3.2 Technical Approach

In this section we describe our approach to attacking white-box face-recognition systems. First, we provide the details of the state-of-the-art face-recognition systems we attack (Sec. 3.2.1). Then, we present how we build on previous work [66, 157, 206] to formalize how an attacker can achieve impersonation, and how to generalize the approach for dodging (Sec. 3.2.2). Finally, we discuss how the attacker can conceptually and formally tweak her objective to enable the physical realizability of the attack (Sec. 3.2.3).

We used MatConvNet [211], an neural networks toolbox for MATLAB, to train and run face-recognition DNNs, and to test our attacks. An implementation can be found online at: `https://github.com/mahmoods01/accessorize-to-a-crime`.

3.2.1 White-Box DNNs for Face Recognition

Using ML models and particularly DNNs, researchers have developed face-recognition systems that can outperform humans in their ability to recognize faces [81]. Parkhi et al. developed a 39-layer DNN for face recognition and verification that achieves state-of-the-art performance (and outperforms humans) [159] on the Labeled Faces in the Wild (LFW) [81] challenge, a benchmark for testing face-recognition systems' ability to classify images taken under unconstrained conditions.

To demonstrate dodging and impersonation we use three DNNs: First, we use the DNN developed by Parkhi et al.,[1] which we refer to as *VGG2622*. Second, we trained two DNNs (termed VGG10 and VGG143) based on the structure of Parkhi et al.'s to recognize celebrities as well as people who were available to us in person for testing real attacks. The purpose of using VGG10 and VGG143 is to test the physically realized versions of our attacks, which require people to wear glasses designed by our algorithm. Testing dodging requires the subjects to be known to the classifier (otherwise they would always be misclassified). Similarly, testing impersonation is more realistic (and more difficult for the attacker) if the DNN is able to recognize the impersonator. The DNNs we use in this work can be conceptualized as differentiable functions that map input images to probability distributions over classes. An image is counted as belonging to the class that receives the highest probability, optionally only if that probability exceeds a predefined threshold.

VGG2622 Parkhi et al. trained VGG2622 to recognize 2622 celebrities. They used roughly 1000 images per celebrity for training, for a total of about 2.6M images. Even though they used much less data for training than previous work (e.g., Google used 200M [183]), their DNN still achieves 98.95% accuracy, comparable to other state-of-the-art DNNs [81].

VGG10 and VGG143 Using VGG2622 for physical realizability experiments is not ideal, as it was not trained to recognize people available to us for testing physically realized attacks. Therefore, we trained two additional DNNs.

VGG10 was trained to recognize ten subjects: five people from our lab (the author of the thesis, two collaborators, and two additional researchers who volunteered images for training), and five celebrities for whom we picked images from the PubFig image dataset [111]. In total, the training set contained five females and five males of ages 20 to 53 years. The celebrities we used for training were: Aaron Eckhart, Brad Pitt, Clive Owen, Drew Barrymore, and Milla Jovovich.

VGG143 was trained to recognize a larger set of people, arguably posing a tougher challenge for attackers attempting impersonation attacks. In total, VGG143 was trained to recognize 143 subjects: 140 celebrities from PubFig's [111] evaluation set, the author of the thesis, and two collaborators.

We trained VGG10 and VGG143 via *transfer learning*, a traditional procedure to train DNNs from other, pre-trained, DNNs [229]. Transfer learning reduces the need for large amounts of data for training a DNN by repurposing a pre-trained DNN for a different, but related, classification task. This is performed by copying an initial set of layers from the existing DNN, appending new layers to them, and training the parameters of the new layers for the new task. Consequently, the layers copied from the

[1] `http://www.robots.ox.ac.uk/~vgg/software/vgg_face/`

old network serve for feature extraction and the extracted features are fed to the new layers for classification. Previous work has shown that transfer learning is effective in training high-performance DNNs from ones that have already been trained when they perform closely related tasks [229].

We followed the suggestion of Yosinski et al. [229] to train VGG10 and VGG143. More specifically, we used the first 37 layers of VGG2622 for feature extraction. Then, we appended a fully connected layer of neurons followed by a *softmax* layer, and updated the weights in the fully connected layer via the back-propagation algorithm. The neurons in the fully-connected layers serve as linear classifiers that classify the features extracted by VGG2622 into identities, and the *softmax* layer transforms their output into a probability distribution. Training more layers (i.e., more parameters to tune) was prohibitive, as the amount of data we could collect from people available for testing physically realizable attacks was limited. We used about 40 images per subject for training—an amount of images that was small enough to collect, but high enough to train highly performing DNNs. On images held aside for testing, VGG10 achieved classification accuracy of 97.43%, and VGG143 achieved accuracy of 96.75%.

Similarly to Parkhi et al., we used 2d affine alignment to align face images to a canonical pose at the input of the DNNs [159]. Additionally, we resized input images to 224×224 (the input dimensionality of the DNNs we use).

3.2.2 Attacking White-Box Face-Recognition Systems

Following Parkhi et al. we adopt the cross-entropy loss ($Loss_{ce}$) score to measure the correctness of classifications [159]. Formally, given an input x of class c_x that is classified as $\mathbb{F}(x)$ (a vector of probabilities), $Loss_{ce}$ is defined as follows:

$$Loss_{ce}(\mathbb{F}(x), c_x) = -\log(\langle \mathbb{H}_{c_x}, \mathbb{F}(x) \rangle)$$

where $\langle \cdot, \cdot \rangle$ denotes inner product between two vectors, N is the number of classes, and $\mathbb{H}_c$ is the one-hot vector of class c. It follows that $Loss_{ce}$ is lower when the DNN classifies x correctly, and higher when the classification is incorrect. We use $Loss_{ce}$ to define the impersonation and dodging objectives of attackers.

Impersonation An attacker who wishes to impersonate a target t needs to find how to perturb the input x via an addition r to maximize the probability of class c_t. Szegedy et al. defined this as minimizing the distance between $\mathbb{F}(x+r)$ and the c_t (see Sec. 2.1). Similarly, we define the optimization problem to be solved for impersonation as:

$$\underset{r}{\operatorname{argmin}}\, Loss_{ce}(\mathbb{F}(x+r), c_t)$$

In other words, we seek to find a modification r of image x that will minimize the distance of the modified image to the target class c_t.

Dodging In contrast to an attacker who wishes to impersonate a particular target, an attacker who wishes to dodge recognition is only interested in not being recognized as herself. To accomplish this goal, the attacker needs to find how to perturb the input x to minimize the probability of the class c_x. Such a perturbation r would maximize the value of $Loss_{ce}(\mathbb{F}(x+r), c_x)$. To this end, we define the optimization problem for

dodging as:

$$\underset{r}{\text{argmin}} \big(- Loss_{ce}(\mathbb{F}(x+r), c_x) \big)$$

To solve these optimizations, we use the Gradient Descent (GD) algorithm [22]. GD is guaranteed to find a global minimum only when the objective is convex, but in practice often finds useful solutions regardless. In a nutshell, GD is an iterative algorithm that takes an initial solution to r and iteratively refines it to minimize the objective. GD iteratively evaluates the gradient, g, and updates the solution by $r = r - \alpha g$, for a positive value α (i.e., r is updated by "stepping" in the direction of the steepest descent). GD stops after convergence (i.e., changes in the objective function become negligible) or after a fixed number of iterations. In our work, we fix the number of iterations.

3.2.3 Facilitating Physical Realizability

Accomplishing impersonation or dodging in a digital environment does not guarantee that the attack will be physically realizable, as our experiments in Sec. 3.3.1 show. Therefore, we take steps to enable physical realizability. The first step involves implementing the attacks purely with facial accessories (specifically, eyeglass frames), which are physically realizable via 3d- or even 2d-printing technologies. The second step involves tweaking the mathematical formulation of the attacker's objective to focus on adversarial perturbations that are *a)* robust to small changes in viewing condition; *b)* smooth (as expected from natural images); and *c)* realizable by affordable printing technologies. In what follows, we describe the details of each step.

Utilizing Facial Accessories While an attacker who perturbs arbitrary pixels that overlay her face can accomplish impersonation and dodging attacks, the perturbations may be impossible to implement successfully in practice (see Sec. 3.3.1). To address this, we utilize perturbed facial accessories (in particular, eyeglass frames) to implement the attacks. One advantage of facial accessories is that they can be easily implemented. In particular, we use a commodity inkjet printer (Epson XP-830) to print the front plane of the eyeglass frames on glossy paper, which we then affix to actual eyeglass frames when physically realizing attacks. Moreover, facial accessories, such as eyeglasses, help make attacks plausibly deniable, as it is natural for people to wear them.

Unless otherwise mentioned, in our experiments we use the eyeglass frames depicted in Fig. 3.1. These have a similar design to frames called "geek" frames in the eyewear industry.[2] We select them because of the easy availability of a digital model. After alignment, the frames occupy about 6.5% of the pixels of the 224×224 face images, similarly to real frames we have tested. This implies that the attacks perturb at most 6.5% of the pixels in the image.

To find the color of the frames necessary to achieve impersonation or dodging we first initialize their color to a solid color (e.g., yellow). Subsequently, we render the frames onto the image of the subject attempting the attack and iteratively update

[2]E.g., see: `https://goo.gl/Nsd20I`

Figure 3.1: The eyeglass frames used to fool face-recognition systems (before texture perturbation). By Clker-Free-Vector-Images / CC0 / `https://goo.gl/3RHKZA`.

their color through the GD process. In each iteration, we randomize the position of the frames around a reference position by moving them by up to three pixels horizontally or vertically, and by rotating them up to four degrees. The rationale behind doing so is to craft adversarial perturbations that are tolerant to slight movements that are natural when physically wearing the frames.

Enhancing Perturbations' Robustness Due to varying imaging conditions, such as changes in expression and pose, two images of the same face are unlikely to be exactly the same. As a result, to successfully realize the attacks, attackers need to find perturbations that are independent of the exact imaging conditions. In other words, an attacker would need to find perturbations that generalize beyond a single image, and can be used to deceive the face-recognition systems into misclassifying many images of the attacker's face.

Thus far, the techniques to find perturbations were specific to a given input. To enhance the generality of the perturbations, we look for perturbations that can cause any image in a *set of inputs* to be misclassified. To this end, an attacker collects a set of images, X, and finds a single perturbation that optimizes her objective for every image $x \in X$. For impersonation, we formalize this as the following optimization problem (dodging is analogous):

$$\underset{r}{\operatorname{argmin}} \sum_{x \in X} Loss_{ce}(\mathbb{F}(x+r), l)$$

Enhancing Perturbations' Smoothness Natural images (i.e., those captured in reality) are known to comprise smooth and consistent patches, where colors change only gradually within patches [130]. Therefore, to enhance plausible deniability, it is desirable to find perturbations that are smooth and consistent. In addition, due to sampling noise, extreme differences between adjacent pixels in the perturbation are unlikely to be accurately captured by cameras. Consequently, perturbations that are non-smooth may not be physically realizable.

To maintain the smoothness of perturbations, we update the optimization to account for minimizing *total variation* (TV) [130]. For a perturbation r, $TV(r)$ is defined as:

$$TV(r) = \sum_{i,j} \left((r_{i,j} - r_{i+1,j})^2 + (r_{i,j} - r_{i,j+1})^2 \right)^{\frac{1}{2}}$$

where $r_{i,j}$ are is a pixel in r at coordinates (i, j). $TV(r)$ is low when the values of adjacent pixels are close to each other (i.e., the perturbation is smooth), and high otherwise. Hence, by minimizing $TV(r)$ we improve the smoothness of the perturbed image and improve physical realizability.

Enhancing Perturbations' Printability The range of colors that devices such as printers and screens can reproduce (the color *gamut*) is only a subset of the $[0,1]^3$ RGB color space. Thus, to be able to successfully use a printer to realize adversarial perturbations, it is desirable to craft perturbations that are comprised mostly of colors reproducible by the printer. To find such perturbations, we define the *non-printability score* (*NPS*) of images to be high for images that contain unreproducible colors, and low otherwise. We then include minimizing the non-printability score as part of our optimization.

Let $P \subset [0,1]^3$ be the set of printable RGB triplets. We define the *NPS* of a pixel $\hat{p}$ as:

$$NPS(\hat{p}) = \prod_{p \in P} |\hat{p} - p|$$

If $\hat{p}$ belongs to P, or if it is close to some $p \in P$, then $NPS(p)$ will be low. Otherwise, $NPS(p)$ will be high. We intuitively generalize the definition of *NPS* of a perturbation as the sum of *NPS*s of all the pixels in the perturbation.

In practice, to approximate the color gamut of a printer, we print a color palette that comprises a fifth of the RGB color space (with uniform spacing). Subsequently, we capture an image of the palette with a camera to acquire a set of RGB triplets that can be printed. As the number of unique triplets is large ($\geq 10K$), the computation of *NPS* becomes computationally expensive. To this end, we quantize the set of colors to a set of 30 RGB triplets that have a minimal variance in distances from the complete set, and use only those in the definition of *NPS*; this is an optimization that we have found effective in practice.

In addition to having limited gamut, printers do not reproduce colors faithfully, i.e., the RGB values of a pixel requested to be printed do not correspond exactly to the RGB values of the printed pixel. To allow us to realize perturbations with high fidelity, we create a map m that maps colors (i.e., RGB triplets) requested to be printed to colors that are actually printed. We then utilize this map to realize perturbations with high fidelity. In particular, to print a perturbation as faithfully as possible, we replace the value p of each pixel in the perturbation with $\tilde{p}$, such that the printing error, $|p - m(\tilde{p})|$, is minimized. This process can be thought of as manual color management [106].

3.3 Evaluation

We separately evaluate attacks that take place purely in the digital domain (Sec. 3.3.1) and physically realized attacks (Sec. 3.3.2).

3.3.1 Digital-Environment Experiments

We first discuss experiments that assess the difficulty of deceiving face-recognition systems in a setting where the attacker can manipulate the digital input to the system, i.e., modify the images to be classified on a per-pixel level. Intuitively, an attacker who cannot fool face-recognition systems successfully in such a setting will also struggle to do so physically in practice.

Exp. #	*Area perturbed*	*Goal*	*Model*	*# Attackers*	*Success rate*
1	Entire face	Dodging	VGG2622	20	100.00%
2	Entire face	Impersonation	VGG2622	20	100.00%
3	Eyeglass frames	Dodging	VGG2622	20	100.00%
4	Eyeglass frames	Dodging	VGG10	10	100.00%
5	Eyeglass frames	Dodging	VGG143	20	100.00%
6	Eyeglass frames	Impersonation	VGG2622	20	91.67%
7	Eyeglass frames	Impersonation	VGG10	10	100.00%
8	Eyeglass frames	Impersonation	VGG143	20	100.00%

Table 3.1: A summary of the digital-environment experiments attacking VGG2622, VGG10, and VGG143 under the white-box scenario. In each attack we used three images of the subject that we sought to misclassify; the reported success rate is the mean success rate across those images.

Experiment Description We ran eight dodging and impersonation attacks (see Sec. 3.2.2) on the white-box DNNs presented in Sec. 3.2.1. Between experiments, we varied (1) the attacker's goal (dodging or impersonation), (2) the area the attacker can perturb (the whole face or just eyeglass frames that the subject wears), and (3) the DNN that is attacked (VGG2622, VGG10, or VGG143). The frames we used are depicted in Fig. 3.1.

We measured the attacker's success (*success rate*) as the fraction of attempts in which she was able to achieve her goal. For dodging, the goal is merely to be misclassified, i.e., the DNN, when computing the probability of the image belonging to each target class, should find that the most probable class is one that does not identify the attacker. For impersonation, the goal is to be classified as a specific target, i.e., the DNN should compute that the most probable class for the input image is the class that identifies the target. In impersonation attempts, we picked the targets randomly from the set of people the DNN recognizes.

To simulate attacks on VGG2622 and VGG143, we chose at random 20 subjects for each experiment from the 2622 and 143 subjects, respectively, that the DNNs were trained on. We simulated attacks on VGG10 with all ten subjects it was trained to recognize. To compute statistics that generalize beyond individual images, we performed each attack (e.g., each attempt for subject X to impersonate target Y) on three images of the subject and report the mean success rate across those images. We ran the gradient descent process for at most 300 iterations, as going beyond that limit has diminishing returns in practice. A summary of the experiments and their results is presented in Table 3.1.

Experiment Results In experiments 1 and 2 we simulated dodging and impersonation attacks in which the attacker is allowed to perturb any pixel on her face. The attacker successfully achieved her goal in all attempts. Moreover, the adversarial examples found under these settings are likely imperceptible to humans (similar to adversarial examples found by Szegedy et al. [206]): the mean perturbation of a pixel

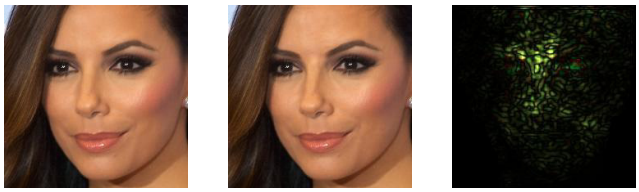


Figure 3.2: A dodging attack by perturbing an entire face. Left: an original image of actress Eva Longoria (by Richard Sandoval / CC BY-SA / cropped from `https://goo.gl/7QUvRq`). Middle: A perturbed image for dodging. Right: The applied perturbation, after multiplying the absolute value of pixels' channels $\times 20$.

Figure 3.3: An impersonation using frames. Left: Actress Reese Witherspoon (by Eva Rinaldi / CC BY-SA / cropped from `https://goo.gl/a2sCdc`). Image classified correctly with probability 1. Middle: Perturbing frames to impersonate (actor) Russell Crowe. Right: The target (by Eva Rinaldi / CC BY-SA / cropped from `https://goo.gl/AO7QYu`).

that overlaid the face was 1.7, with standard deviation 0.93. An example is shown in Fig. 3.2.

We believe that the types of attacks examined in experiments 1 and 2 are far from practical. Because the perturbations are too subtle and lack structure, they may be too complicated to physically realize, e.g., it is likely to be impossible to modify a human face in exactly the way required by the attack. Moreover, the amount by which pixels are perturbed is often much smaller than the error that occurs when printing a perturbation and then capturing it with a scanner or a camera. Our experiments show that the average error per pixel in a perturbation when printed and scanned is 22.9, and its standard deviation is 38.22. Therefore, even if the attacker can successfully realize the perturbation, she is still unlikely to be able to deceive the system.

To this end, in experiments 3–8 we simulated dodging and impersonation attacks in which we perturbed only eyeglass frames of eyeglasses worn by each subject. With the exception of experiment 6, the attacker was able to dodge recognition or impersonate targets in all attempts. In experiment 6 impersonators succeeded in about 91.67% of their attempts to fool VGG2622 using perturbed eyeglasses. We hypothesize that due to the large number of classes VGG2622 recognizes, the image space between some attacker-target pairs is occupied by other classes. Therefore, impersonating these targets requires evading several classification boundaries, making impersonation attacks more difficult. Fig. 3.3 shows an example of a successful impersonation attempt using eyeglass frames.

3.3.2 Physical-Realizability Experiments

As discussed in Sec. 3.2.3, to physically realize an attack, we utilize a set of perturbed eyeglass frames, ensure that the perturbation is smooth and effective for misclassifying more than one image, and enhance the reproducibility of the perturbation's colors by the printing device. To achieve these goals and impersonate a target t, an attacker finds a perturbation by solving the following optimization problem:

$$\underset{r}{\operatorname{argmin}} \left(\left(\sum_{x \in X} Loss_{ce}(x + r, c_t) \right) + \kappa_1 \cdot TV(r) + \kappa_2 \cdot NPS(r) \right)$$

where κ_1 and κ_2 are constants for balancing the objectives and X is a set of images of the attacker. The formulation for dodging is analogous.

In this section we report on experiments for evaluating the efficacy of this approach in fooling VGG10 and VGG143 under semi-controlled imaging conditions.

Experiment Description Two collaborators on the projects (S_A, a 41-year-old white male, and S_B, a 24-year-old South Asian female, at the time of the experiment), and the author of the thesis (S_C, a 24-year-old Middle Eastern male at the time of the experiment), whom VGG10 and VGG143 were trained to recognize. For each subject we attempted two dodging attacks and two impersonation attacks—one of each type of attack on each of VGG10 and VGG143. The targets in impersonation attacks were randomly selected (see Table 3.2).

We collected images of the subjects using a Canon T4i camera. To prevent extreme lighting variations, we collected images in a room without exterior windows. Subjects stood a fixed distance from the camera and were told to maintain a neutral expression and to make slight pose changes throughout the collection. While these collection conditions are only a subset of what would be encountered in practice, we believe they are realistic for some scenarios where face-recognition system technology is used, e.g., within a building for access control.

For each subject, we collected 30–50 images in each of five sessions. In the first session, we collected a set of images that was used for generating the attacks (referred to as the set X in the mathematical representation). In this session, the subjects did not wear the eyeglass frames. In the second and third sessions, the subjects wore eyeglass frames to attempt dodging against VGG10 and VGG143, respectively. In the fourth and the fifth sessions, the subjects wore frames to attempt impersonation against VGG10 and VGG143.

We physically realized attacks by printing the eyeglass frames with an Epson XP-830 printer on Epson Glossy photo paper. Realizing these attacks is cheap and affordable; the approximate cost for printing an eyeglass frame is \$0.22. Once printed, we cut out the frames and affixed them to the frames of an actual pair of eyeglasses. Examples of realizations are shown in Fig. 3.4.

To find the perturbation, the parameters κ_1 and κ_2 in the optimization were set to 0.15 and 0.25, respectively. The computational overhead of mounting attacks prohibited a broad exploration of the parameter space, but we found these values effective in practice. In addition, we limited the number of iterations of the GD process to 300.

DNN	Subject	Dodging *SR*	Dodging $E(p(c_x))$	Impersonation *Target*	Impersonation *SR*	Impersonation *HC*	Impersonation $E(p(c_t))$
VGG10	S_A	100.00%	0.01	Milla Jovovich	87.87%	48.48%	0.78
	S_B	97.22%	0.03	S_C	88.00%	75.00%	0.75
	S_C	80.00%	0.35	Clive Owen	16.13%	0.00%	0.33
VGG143	S_A	100.00%	0.03	John Malkovich	100.00%	100.00%	0.99
	S_B	100.00%	<0.01	Colin Powell	16.22%	0.00%	0.08
	S_C	100.00%	<0.01	Carson Daly	100.00%	100.00%	0.90

Table 3.2: A summary of the physical realizability experiments. To the left, we report the DNN attacked and the identity of the subjects (the attackers in the simulation). When not attempting to fool both DNNs, the subjects were originally classified correctly with mean probability >0.85. SR is the success rate. HC is the success rate when the attacker's image is misclassified as the target with high confidence (i.e., above a threshold set to balance the security and the usability of the face-recognition system). $E(p(c))$ is the mean (expected) probability of the class when classifying all images (c_x is the correct class, c_t is the target class). Results for S_C when attacking VGG10 were achieved with glasses that occupy 10% of the area of the image being classified; results for the other experiments were achieved with glasses that occupy 6% of the image.

Experiment Results To evaluate VGG10 and VGG143 in a non-adversarial setting, we classified the non-adversarial images collected in the first session. All the face images of the three subjects were classified correctly. The mean probability of the correct class across the classification attempts was above 0.85, which implies that naive attempts at impersonation or dodging are highly unlikely to succeed. In contrast, our experiments showed that an attacker who intentionally attempts to deceive the system will usually succeed. Table 3.2 summarizes these results. Fig. 3.4 presents examples of successful dodging attacks and impersonations.

All three subjects successfully dodged face recognition by wearing perturbed eyeglass frames. When wearing frames for dodging, as shown in Fig. 3.4a, all of S_A's images collected in the second and the third sessions were misclassified by VGG10 and VGG143, respectively. When dodging, the mean probability VGG10 assigned to the S_A's class dropped remarkably from 1 to 0.01. Similarly, the mean probability assigned to c_{S_A} by VGG143 dropped from 0.85 to 0.03. Thus, the dodging attempts made it highly unlikely that the DNNs would output c_{S_A}, the correct classification result. S_B was able to dodge recognition in 97.22% of the attempts against VGG10 and in 100% of the attempts against VGG143. When classifying S_B's images, the mean probability assigned by the DNNs to c_{S_B} became ≤ 0.03 as a result of the attack, also making it unlikely to be the class assigned by the DNN.

S_C's attempts in dodging recognition against VGG143 were also successful: all his images collected in the third session were misclassified, and the probability assigned to c_{S_C} was low (<0.01). However, when wearing a perturbed version of the frames shown in Fig. 3.1; none of S_C's images collected as part of the second session misled VGG10. We conjecture that because S_C was the only subject who wore eyeglasses among the

(a) (b) (c) (d)

Figure 3.4: Examples of successful impersonation and dodging attacks. Fig. (a) shows S_A (top) and S_B (bottom) dodging against VGG10. Fig. (b)–(d) show impersonations. Impersonators carrying out the attack are shown in the top row and corresponding impersonation targets in the bottom row. Fig. (b) shows S_A impersonating Milla Jovovich (by Georges Biard / CC BY-SA / cropped from `https://goo.gl/GlsWlC`); (c) S_B impersonating S_C; and (d) S_C impersonating Carson Daly (by Anthony Quintano / CC BY / cropped from `https://goo.gl/VfnDct`).

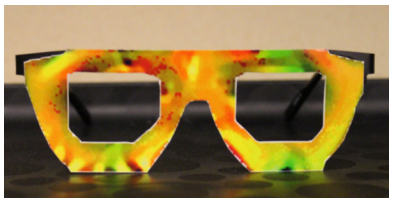


Figure 3.5: The eyeglass frames used by S_C for dodging recognition against VGG10.

subjects used for training VGG10, c_{S_C} became a likely class when classifying images showing people wearing eyeglasses. Therefore, it became particularly hard for S_C to fool VGG10 by perturbing only a small area of the image. Nevertheless, by increasing the size of the frames such that they occupied 10% of the area of the aligned image (frames shown in Fig. 3.5) it became possible for S_C to achieve physically realizable dodging at the cost of decreased inconspicuousness. Using the larger frames, S_C was able to dodge recognition in 80% of the images (mean probability of c_{S_C} dropped to 0.35).

To simulate impersonation attempts, each subject was assigned two random targets: one for fooling VGG10 and one for fooling VGG143. S_A was assigned to impersonate Milla Jovovich, a 40-year-old white female, and John Malkovich, a 62-year-old white male; S_B was assigned to impersonate S_C, and Colin Powell, a 79-year-old white male; and S_C was assigned to impersonate Clive Owen, a 51-year-old male, and Carson Daly, a 43-year-old male.

Both of S_A's impersonation attempts were successful: 87.87% of his images collected in the fourth session were misclassified by VGG10 as Milla Jovovich (the mean probability of the target was 0.78), and all the images collected in the fifth session were

misclassified as John Malkovich (mean probability of 0.99). In contrast, S_B and S_C had mixed success. On the one hand, S_B misled VGG10 by successfully impersonating S_C in 88% of her attempts (the mean probability of the target was 0.75), and S_C misled VGG143 into misclassifying him as Carson Daly in all of his attempts (mean probability of 0.99). On the other hand, they were able to successfully impersonate Colin Powell and Clive Owen[3] (respectively) only in about one of every six attempts. This success rate may be sufficient against, e.g., access-control systems that do not limit the number of recognition attempts, but may not be sufficient against systems that limit the number of attempts or in surveillance applications. We hypothesize that some targets are particularly difficult for some subjects to impersonate. We also believe, however, that even in those cases further refinement of the attacks can lead to greater success than we have so far measured.

In practice, to tune the security of a face-recognition system, system operators may set a minimum threshold that the maximum probability in the DNN's output will have to exceed for a classification result to be accepted. Such a threshold balances security and usability: If the threshold is high, misclassification occurs less often, but correct classifications may be rejected, thus harming usability. On the other hand, for low thresholds, correct classifications will be accepted most of the time, but misclassifications are more likely to occur, which would harm security. Therefore, system operators usually try to find a threshold that balances the usability and the security of the deployed face-recognition system. Using the test data, we found that by setting the threshold to 0.85, the false acceptance rate of VGG10 became 0, while true acceptance became 92.31%. This threshold strikes a good balance between usability and security [87]; by using it, false acceptance (of zero-effort impostors) never occurs, while true acceptance remains high. Following a similar procedure, we found that a threshold of 0.90 achieved a reasonable tradeoff between security and usability for VGG143; the true acceptance rate became 92.01% and the false acceptance rate became 4e−3. Attempting to decrease the false acceptance rate to 0 reduced the true acceptance rate to 41.42%, making the face-recognition system unusable.

Using thresholds changes the definition of successful impersonation: to successfully impersonate the target t, the probability assigned to c_t must exceed the threshold. Evaluating the previous impersonation attempts under this definition, we found that success rates generally decreased but remained high enough for the impersonations to be considered a real threat (see Table 3.2). For example, S_B's success rate when attempting to fool VGG10 and impersonate S_C decreased from 88.00% without threshold to 75.00% when using a threshold.

Time Complexity The DNNs we use in this work are large, e.g., the number of connections in VGG10, the smallest DNN, is about 3.86e8. Thus, the main overhead when solving the optimization problem via GD is computing the derivatives of the DNNs with respect to the input images. For N_I images used in the optimizations and N_C connections in the DNN, the time complexity of each GD iteration is $\mathcal{O}(N_I * N_C)$. In practice, when using about 30 images, one iteration of GD on a 2015 MacBook Pro (equipped with 16GB of memory and a 2.2GHz Intel i7 CPU) takes about 52.72

[3]Similarly to dodging, S_C required larger frames to impersonate the target.

seconds. Hence, running the optimization up to 300 iterations may take about 4.39 hours.

3.4 Extension to Black-box Models

So far we have examined attacks in a white-box setting, where the adversary has access to the model she is trying to deceive. In this section we demonstrate how similar attacks can be applied in a black-box scenario. In such a scenario, the adversary would typically have access only to an oracle O which outputs a result for a given input and allows a limited number of queries. The threat model we consider here is one in which the adversary has access only to the oracle.

We next briefly describe a commercial face-recognition system that we use in our experiments (Sec. 3.4.1), and then describe and evaluate preliminary attempts to carry out impersonation attacks in a black-box setting (Sec. 3.4.2–3.4.3).

3.4.1 Face++: A Commercial Face-Recognition System

Face++ is a cross-platform commercial state-of-the-art face-recognition system that is widely used by applications for facial recognition, detection, tracking, and analysis [235]. It has been shown to achieve accuracy over 97.3% on LFW [58]. Face++ allows users to upload training images and labels and trains a face-recognition system that can be queried by applications. Given an image, the output from Face++ is the top three most probable classes of the image along with their confidence scores. Face++ is marketed as "face recognition in the cloud." Users have no access to the internals of the training process and the model used, nor even to a precise explanation of the meaning of the confidence scores. Face++ is rate-limited to 50,000 free queries per month per user.

To train the Face++ model, we used the same training data used for VGG10 in Sec. 3.2.1 to create a 10-class face-recognition system.

3.4.2 Impersonation Attacks on Face++

The goal of our black-box attack is for an adversary to alter an image to which she has access so that it is misclassified. We attempted dodging attacks with randomly colored glasses and found that it worked immediately for several images. Therefore, in this section we focus on the problem of impersonation from a given *source* to a *target*. We treat Face++ as an example black-box face-recognition system, with its query function modeled as the oracle $O(x)$. The oracle returns *candidates*, an ordered list of three classes numbered from 1 to 3 in decreasing order of confidence.

Our algorithm for attacking Face++ uses particle swarm optimization (PSO) [52] as a subroutine. We begin by summarizing this technique.

Particle Swarm Optimization (PSO) Particle swarm optimization is a heuristic and stochastic algorithm for finding solutions to optimization problems by mimicking the behavior of a swarm of birds [52]. It iterates on a set of candidate solutions, called *particles*, and collectively called a *seed*, that it updates based on the evaluation of an

objective function. Each particle is a candidate solution and occupies a *position* in the solution space. The *value* of a particle is the result of evaluating the objective function on that particle's position in the solution space. In each iteration, each particle is updated by adding a *velocity* to its position. The velocity is a weighted and randomized linear combination of the distance between (1) the current position of the particle and its best position thus far (P_{best}) and (2) the current position of the particle and the best position taken by *any* particle thus far (G_{best}), where "best" indicates that the objective function evaluates to the smallest value. Once a termination criterion is met, G_{best} should hold the solution for a local minimum.

We choose this over other black-box optimization methods such as surrogate models [21]—which require that the adversary has the training data and enough computational resources to train an effective surrogate—and genetic algorithms [17]—which though similar to PSO are much less computationally efficient [74, 168].

Since Face++ only returns the top three classes, PSO can make progress if *target* is in the top three results reported for any particle in the first iteration. However, when *target* is not in the top three for any set of initial solutions, the algorithm does not get any feedback on appropriate directions. To address this, we implement an algorithm we call *recursive impersonation* (Alg. 1). The goal of the algorithm to is reach the final target by attempting multiple intermediate impersonations on varying targets in successive runs of PSO. This is done in the hope that attempting an intermediate impersonation will move the swarm away from the previous solution space and toward candidate solutions that may result in the *target* being returned in the top three classes. If the target does not initially appear in the candidate list that results from querying the oracle with the original image, we select the class with the second highest confidence to be the intermediate target. On subsequent PSO runs, the most commonly occurring class labels that have not yet been used as targets become the new intermediate targets.

In our implementation, we modify the PSO subroutine to globally keep track of all the particles used in the last iteration of PSO as well as all particles throughout all PSO iterations for which invoking the oracle resulted in *target* appearing in the candidate list. The invoking algorithm has access to these saved particles and uses them in order to select a new intermediate impersonation target or to provide a seed for the next impersonation attempt.

On each run, PSO aims to minimize an objective function defined by $f(x + r)$, where r is the perturbation applied to the image x. $f(\cdot)$ is computed based on the output from the oracle O. The value of this objective at every particle is then used to move the swarm in a new direction during the next iteration of PSO. We experimented with several definitions of $f(\cdot)$. In practice, we found the following to be the most effective:

$$f(x) = \begin{cases} rank \cdot \frac{score_{top}}{score_{target}} & \text{if } target \in candidates \\ maxObjective & \text{if } target \notin candidates \end{cases}$$

The function uses the input x to query O for the *candidates*. The variable $score_{top}$ denotes the confidence score of the top-ranked item in *candidates*. If the *target* is in *candidates* then $score_{target}$ is its confidence score and *rank* its rank in *candidates*. When *target* is successfully impersonated, $f(\cdot)$ receives its minimal value of 1. If *target* is not in the top three candidates, the function should evaluate to *maxObjective*, which we

set to be a sufficiently large value to indicate a result far less optimal than P_{best} or G_{best}.

Algorithm 1: Recursive Impersonation

Initialize $epoch = 0$, $numParticles$, $epochs_{max}$ and $seed$.
Set $candidates = O(image_{original})$.
if $target \in candidates$ **then** $target_{current} = target$;
else $target_{current}$ = 2nd most probable class of $candidates$;
while $epoch \leq epoch_{max}$ **do**
    Run PSO subroutine with $target_{current}$ and $seed$.
    if any particle impersonated $target$ during PSO **then**
        solution was found. exit.
    else if $target \in candidates$ of any query during PSO **then**
        $target_{current} = target$. Clear $seed$.
        $seed \supseteq$ particles that produced this candidate from the current PSO run.
    else
        if new candidate emerges from current PSO run **then**
            $target_{current}$ = new candidate. Clear $seed$.
            $seed \supseteq$ particles that produced this candidate from the current PSO run.
        else
            no solution was found. exit.
    $epoch = epoch + 1$

3.4.3 Results

Experiment Description To evaluate our methodology, we picked four $\{source, target\}$ pairs from among the subjects on which Face++ was trained. Two of the pairs were chosen at random. The other two pairs were chosen as challenging impersonations by making sure that the target was not one of the top three classes reported by Face++ for the source image. To make the attack realistic in terms of possibility of physical realization, we restricted the perturbations to a pair of glasses. We did not attempt to physically realize the attack, however.

We ran our experiments with 25 particles. The particles were initialized by randomly generating 25 distinct pairs of glasses with smooth color variations. We ran the algorithm for a maximum of 15 epochs or attempts, and set the iteration limit of PSO to 50 and $maxObjective$ to 50. We also assigned weights in computing the velocity such that a higher weight was given to G_{best} than P_{best}.

Experiment Results The results from the experiments are shown in Table 3.3. All attempted impersonations succeeded. Noteworthy is the low number of queries needed for the attacks, which shows that rate-limiting access to services will not always stop an online attack.

Source	*Target*	*Success rate*	*Avg. # queries*
Clive Owen	S_A	100%	109
Drew Barrymore	S_B	100%	134
S_D	S_C	100%	25
S_D	Clive Owen	100%	59

Table 3.3: Results of four attempted impersonation attacks, each run three times. S_A–S_C are the same subjects from Sec. 3.3.2. S_D is a 33-year-old Asian female. Each attempt had a different (randomized) initial seed and velocities. Number of queries is the total number of queries made of the oracle in the PSO iterations.

3.5 Extension to Face Detection

In this section, we show how to generalize the basic approach presented in Sec. 3.2.2 to achieve *invisibility* to facial biometric systems. In an invisibility attack, an adversary seeks to trick an ML system not into misclassifying one person as another, but into simply failing to detect the presence of a person.

We examine this category of attacks for two reasons. First, most ML systems for identifying faces have two phases: detecting the presence of a face and then identifying the detected face. The detection phase is typically less closely tied to training data than the recognition phase. Hence, techniques to circumvent detection have the potential to apply more broadly across multiple systems.

Second, avoiding detection corresponds naturally to one type of motivation—the desire to achieve privacy. In seeking to achieve privacy, a person may specifically want to avoid causing culpability to be placed on another person. Similarly, a mislabeling of a face might be more likely to arouse suspicion or alert authorities than would the failure to notice the presence of a face at all, as might occur at an airport security checkpoint where faces detected by face-detection systems are confirmed against face images of passengers expected to travel, or faces of people wanted by the authorities.

In this section, we show how to perform invisibility attacks while attempting to maintain plausible deniability through the use of facial accessories. We defer the examination of the physical realizability of these attacks to future work.

3.5.1 The Viola-Jones Face Detector

The Viola-Jones (VJ) face detector was designed with efficiency and accuracy in mind [213]. The key idea to achieve both goals is to use a cascade of classifiers that have an ascending order of complexity. Each classifier is trained to detect the majority of the positive instances (presence of a face) and reject a large number of the negative instances. To detect an object in an image, several sub-windows are taken from the image and are evaluated by the detector. To be detected as a positive example, the sub-window needs to be classified as a positive example by all the classifiers in the cascade. On the other hand, being rejected by one classifier in the cascade results in classifying a sub-window as a negative example. Sub-windows that are rejected by simple classifiers are not further evaluated by the more sophisticated classifiers.

A classifier in the cascade is composed of a combination of *weak classifiers*. A weak classifier i is a simple classifier that outputs one of two possible values, $\tilde{a}_i$ or $\hat{a}_i$, based on one feature value, $f_i(\cdot)$, and a threshold b_i. Given a classifier that is composed of C weak classifiers, its decision function is defined as:

$$\mathit{Classify}(x) = \left(\sum_{i=1}^{C} \left((\tilde{a}_i - \hat{a}_i)(f_i(x) > b_i) + \hat{a}_i \right) \right) > \tau$$

where τ is the passing threshold of the classifier, x is the sub-window, and $f_i(x) > b_i$ evaluates to 1 if true and 0 otherwise.

As explained above, the VJ detector rejects a sub-window in case one of its classifiers rejects it. Thus, to evade detection it is sufficient to fool one cascade stage. Since the trained VJ is an open source (i.e., white-box) classifier [88], to find a minimal perturbation that can be used for evasion, we could potentially adapt and utilize the solution proposed by Szegedy et al. [206]. However, to solve the optimization problem, we need the classification function to be differentiable, which $\mathit{Classify}(x)$ is not. Therefore, we use the *sigmoid* function, *sig* (as is often done in ML [175]), and formulate the optimization problem as:

$$\underset{r}{\operatorname{argmin}} \left(\left(\sum_{i=1}^{C} \left((\tilde{a}_i - \hat{a}_i) \cdot \mathit{sig}(k \cdot (f_i(x+r) - b_i)) + \hat{a}_i \right) - \tau \right) + c|r| \right) \tag{3.1}$$

where k is a positive real number that can be tuned to control the precision of the approximation. With this approximation, we can perform GD to solve the optimization problem.

3.5.2 Experiment Results

By generating a perturbation to evade a specific stage of the detector via the above technique, we are able to learn how to tweak pixel intensities in specific regions to successfully evade the whole cascade. To test this approach, we randomly selected 20 frontal images from the PubFig [111] face dataset, and tested whether each could be permuted to evade detection by fooling the first classifier in the cascade. We generated perturbed images as follows: we limited the perturbation to the area of the face, set c to 0.015 (as we found this to yield smaller perturbations in practice), and then performed line search on k to find the minimal perturbation necessary to evade the classifier, using a Limited BFGS [150] solver to solve the optimization (Eqn. 3.1).

For 19 out of the 20 images it was possible to evade detection. For the images that achieved evasion, the mean perturbation—the aggregate change in the value of the R, G, and B channels—of a pixel that overlays the face was 16.06 (standard deviation 6.35), which is relatively high and noticeable. As Fig. 3.6 shows, in some cases even the minimal perturbation necessary to evade detection required making changes to faces that could draw increased attention.

In another version of the attack, in which we sought to increase both the success rate and plausible deniability, we first added facial accessories specifically selected for their colors and contrast to the image; we then perturbed the image as in the previous

Figure 3.6: An example of an invisibility attack. Left: original image of actor Kiefer Sutherland. Middle: Invisibility by perturbing pixels that overlay the face. Right: Invisibility with the use of accessories.

attack. The accessories we used were: eyeglasses, a blond wig, bright eye contacts, eye blacks, and a winter hat. With this approach, it was possible to evade detection for all 20 face images. In addition, the amount by which each pixel needed to be perturbed dropped remarkably, thus contributing to plausible deniability. The mean perturbation of a pixel to avoid detection was 3.01 (standard deviation 2.12). An illustration of evasion attacks on VJ that utilize accessories is shown in Fig. 3.6.

3.6 Discussion

Here we discuss the implications and the limitations of the work presented in this chapter.

3.6.1 Implications

As our reliance on technology increases, we sometimes forget that it can fail. In some cases, failures may be devastating and risk lives [170]. The work presented in this chapter, as well as prior work on practical attacks on ML systems (e.g., [120, 200]), show that the introduction of ML to systems, while bringing benefits, increases the attack surface of these systems. Therefore, we should be careful when integrating ML algorithms into safety- or security-critical systems.

In this chapter we show that face-recognition systems are vulnerable to a new type of attacks: inconspicuous and physically realizable attacks that lead to dodging or impersonation. Such attacks can be especially pernicious as they can resist cursory investigation (at the least) and can provide attackers with plausible deniability. While we demonstrate the efficacy of these attacks on fooling one kind of DNN architecture in the face-recognition domains, similar approaches can apply to other domains. In fact, the techniques demonstrated in this chapter serve as the basis to practical attacks in the street-sign and object-recognition domains (e.g., [11, 57]).

3.6.2 Limitations

The variations in imaging conditions that we investigate in this chapter are narrower than can be encountered in practice. For instance, we controlled lighting by taking images in a room that does not have external windows. These conditions are applicable to some practical cases (e.g., a face-recognition system deployed within a building).

However, other practical scenarios are more challenging, and effective attacks may have to be tolerant to a larger range of imaging conditions. For example, an attacker may not be able to control the lighting or her distance from the camera when a face-recognition system is deployed in the street for surveillance purposes. In Chap. 5, we show that physically realized attacks can be effective under a broader set of imaging conditions than is considered in this chapter (e.g., even when varying lighting conditions markedly).

In addition, the notion of inconspicuousness is subjective, and the only way to measure it adequately would include performing human-subject studies. While we do not present a user study in this chapter, the user study presented in Chap. 5 shows that the attack presented in this chapter sometimes produces eyeglasses that are deemed more realistic by users than actual eyeglasses collected from the web.

3.7 Conclusion

In this chapter, we demonstrated techniques for generating accessories in the form of eyeglass frames that, when printed and worn, can effectively fool state-of-the-art face-recognition systems. Our research builds on research in fooling ML classifiers by perturbing inputs in an adversarial way, but does so with attention to two novel goals: the perturbations must be physically realizable and inconspicuous. We showed that our eyeglass frames enabled subjects to both dodge recognition and to outright impersonate other individuals. We believe that our demonstration of techniques to realize these goals through printed eyeglass frames is both novel and important, and should inform deliberations on the extent to which ML can be trusted in adversarial settings. Finally, we extended our work in two additional directions, first, to so-called black-box face-recognition systems that can be queried but for which the internals are not known, and, second, to defeat state-of-the-art face detection systems.

Chapter 4

Functionality-Preserving Attacks Against Malware Detection

4.1 Introduction

While the majority of work on evasion attacks focuses on the image domain, such attacks are a potential threat to malware detection—a fundamental computer-security problem that is increasingly addressed with the help of ML models (e.g., [7, 107, 165, 207]). In this domain, attackers are interested in altering programs to mislead ML-based malware detectors into misclassifying malicious programs as benign, or vice versa. In doing so, attackers face a non-trivial constraint: in addition to misleading the malware detectors, any alteration of a program must not change its original, intended, functionality. For example, a keylogger altered to evade being detected as malware should still carry out its intended function, including invoking necessary APIs, accessing sensitive files, and connecting to attackers' servers. This constraint is arguably more challenging than ones imposed by other domains (e.g., evading image recognition while making changes imperceptible to humans [206]) as it is less amenable to being encoded into traditional frameworks for generating adversarial examples, and most changes to byte values are likely to break a program's syntax or semantics. In this chapter, we show that the constraint of preserving functionality can be incorporated into the process of generating adversarial examples to fool state-of-the-art deep neural networks (DNNs) for malware detection [107, 165].

Roughly speaking, malware-detection methods can be categorized as dynamic or static [20, 207]. Dynamic methods (e.g., [84]) execute programs to learn behavioral features that can be used for classification. In contrast, static methods (e.g., [7, 107]) classify programs using features that can be computed without execution. While potentially more accurate, dynamic methods are more computationally expensive, and, consequently, less ubiquitously deployed [20, 100]. Therefore, we focus on static methods.

Several attacks have been proposed to generate adversarial examples against DNNs for static malware detection [49, 101, 108, 205]. To fool the DNNs while preserving functionality, these attacks introduce adversarially crafted byte values in regions that do not affect execution (e.g., at the end of programs or between sections). These

attacks can be defended against by masking out or removing the added content before classification (e.g., [110]); we confirm this empirically.

In this chapter we show how binary-diversification tools—tools for transforming programs at the binary level to create diverse variants of the same program—that were originally proposed to defend against code-reuse attacks [105, 158] can be leveraged to evade malware-detection DNNs. While these tools preserve the functionality of programs after transformation by design, they are ineffective at evading malware detection when applied naïvely (e.g., functionality-preserving randomization). To address this, we propose optimization algorithms to guide the transformations of binaries to fool malware-detection DNNs, both in settings where attackers have access to the DNNs' parameters (i.e., white-box) and ones where they have no access (i.e., black-box). The algorithms we propose can produce program variants that often fool DNNs in 100% of evasion attempts. Perhaps most worryingly, we find that the attack samples produced by the algorithms are also often effective at evading commercial malware detectors (in some cases with success rates as high as 85%). Because our attacks transform functional parts of programs, they are particularly difficult to defend against, especially when augmented with methods to deter static and dynamic analyses. We explore potential mitigations to the attacks that we propose (e.g., via preprocessing programs to normalize them before classification [6, 40, 216]), but conclude that attackers may adapt to circumvent these mitigations. This leads us to advocate against relying only on ML-based techniques for malware detection, as is becoming increasingly common [47].

In a nutshell, the contributions of this chapter are as follows:

- We propose a novel functionality-preserving attack against DNNs for malware detection from raw bytes (Sec. 4.2). The attack uses binary-diversification techniques in a novel way to prevent defenses applicable to prior attacks, and is applicable both in white-box and black-box settings.
- We evaluate and demonstrate the effectiveness of the proposed attack in different settings, including against commercial anti-viruses (Sec. 4.3). We also compare our attack with prior attacks, and show that it achieves comparable or higher success rates, while being more challenging to defend against.
- We explore the effectiveness of prior and new defenses against our proposed attack (Sec. 4.4). While several defenses seem promising to defend against specific variants of the attack, we warn against the risk of adaptive attackers.

4.2 Technical Approach

This section discusses the technical approach behind our attack. Before delving into the details, we briefly describe the malware-detection DNNs that we study, provide background on binary-diversification tools that serve as a building block of our attacks, and lay down the threat model.

4.2.1 DNNs for Malware Detection

In this chapter, we mainly study attacks targeting two DNN architectures for detecting malware from the raw bytes of Windows binaries (i.e., executables in Portable Executable format) [107, 165]. The main appeal of these DNNs is that they achieve state-of-the-art performance using automatically learned features, instead of manually crafted features that require tedious human effort (e.g., [7, 86, 102]). In fact, due to their desirable properties, computer-security companies use DNNs similar to the ones that we study (i.e., ones that operate on raw bytes and use a convolution architectures) for malware detection [45]. As these DNNs classify binaries without executing them, they fall under the category of static detection methods [20, 207].

The DNNs proposed by prior work follow standard convolutional architectures similar to the ones used for image classification [107, 165]. Yet, in contrast to image-based classifiers that classify inputs from continuous domains, the malware-detection DNNs classify inputs from discrete domains—byte values of binaries. To this end, the DNNs were designed with initial embedding layers that map each byte in the input to a vector in $\mathbb{R}^8$. Once the input is represented in a real vector space after the embedding, standard convolutional and non-linear operations are performed by subsequent layers.

Fleshman et al. proposed to make malware-detection DNNs more robust by constraining the parameter weights in the last layer to non-negative values [62]. Their approach aims to prevent attackers from introducing additional features to malware to decrease its likelihood of being classified correctly. While this rationale holds for single-layer neural networks (i.e., linear classifiers), DNNs with multiple layers constitute complex functions where the addition of features at the input may correspond to the deletion of features in deep layers. As a result of the misalignment between the threat model and the defense, we found that DNNs trained with this defense are as vulnerable to prior attacks [108] as undefended DNNs. Therefore, we do not consider Fleshman et al.'s defense for the remainder of the chapter.

4.2.2 Binary Diversification

Software diversification is a technique developed to produce diverse binary versions of programs, all with the same functionality, to resist different kinds of attacks, such as memory-corruption, code-injection, and code-reuse attacks [114]. Diversification can be performed at the source-code level (via the development of multiple implementations), at compilation time (e.g., using a multicompiler), or after compilation (by rewriting and randomizing programs' binaries). In this chapter, we build on diversification techniques after compilation, at the binary level, as they have wider applicability (e.g., self-spreading malware can use them to evade detection without having access to the source code [146]), and are more efficient (producing the binary after a transformation does not require recompilation). Nevertheless, we expect that this work can be extended to work with different diversification methods.

There is a large body of work on binary rewriting from the programming-languages, computer-architecture, and computer-security communities (e.g., [72, 104, 105, 133, 158, 181, 218]). Some of the rewriting methods aim to achieve higher-performing code via relatively expensive search through the space of equivalent programs [133, 181].

Other methods significantly increase the size of binaries, or may leave a conspicuous sign that rewriting took place [72, 218]. We build on binary-randomization tools that have little-to-no effect on the size or run time of the randomized binaries, thus helping our attacks remain stealthy [105, 158].

4.2.3 Threat Model

We assume that the attacker has white-box or black-box access to DNNs for malware detection. In the white-box setting, the attacker has access to the DNNs' architectures and weights and is able to efficiently compute the gradients of loss functions with respect to the DNNs' input via forward and backward passes. On the other hand, the attacker in the black-box setting may only query the model with a binary and receive the probability estimate that the binary is malicious.

As is usually the case for inference-time evasion attacks, the weights of the DNNs are fixed and *cannot* be controlled by the attacker (e.g., by poisoning the training data). The attacker uses binary rewriting methods that are challenging to undo to manipulate the raw bytes of binaries and cause misclassification while keeping functionality intact. Attacks may seek to cause malware to be misclassified as benign or benign binaries to be misclassified as malware. The former type of attack may cause malware to circumvent defenses and be executed on a victim's machine. The latter may be useful to induce false positives, which may lead users to turn off or ignore the defenses [76].

We also assume that the binaries are unpacked, as is often the case for static malware-detection methods [20, 107]. Detecting packed binaries and unpacking them are problems orthogonal to ours that have been addressed by other researchers (e.g., [20, 37, 210]). Nonetheless, adversaries may still use our attacks simultaneously with packing: As packed binaries are usually unpacked before being classified by static methods [20], adversaries can use our attacks to modify binaries before packing them so that the binaries would be misclassified once unpacked.

As is standard for ML-based malware detection from raw bytes in particular, and for classification of inputs from discrete domains in general (e.g., [115]), we assume that the first layer of the DNN is an embedding layer. This layer maps each discrete token from the input space to a vector of real numbers via a function $\mathbb{E}(\cdot)$. When computing the DNN's output $\mathbb{F}(x)$ on an input binary x, one first computes the embeddings and feeds them to the subsequent layers. Thus, if we denote the composition of the layers following the embedding by $\mathbb{H}(\cdot)$, then $\mathbb{F}(x) = \mathbb{H}(\mathbb{E}(x))$. While the DNNs we attack contain embedding layers, our attacks conceptually apply to DNNs that do not contain such layers. Specifically, for a DNN function $\mathbb{F}(x) = \ell_{n-1}(\dots \ell_{i+1}(\ell_i(\dots \ell_0(x)\dots))\dots)$ for which the errors can be propagated back to the $(i+1)^{th}$ layer, the attack presented below can be executed by defining $\mathbb{E}(x) = \ell_i(\dots \ell_0(x)\dots)$.

4.2.4 Functionality-Preserving Attack

The attack we propose iteratively transforms a given binary x of class y (y=0 for benign binaries, and y=1 for malware) until misclassification occurs or a maximum number of iterations is reached. To keep the binary's functionality intact, the types of transformations are limited to ones that preserve functionality. (The transformation

types that we consider in this work are detailed below.) In each iteration, the attack picks a transformation type at random for each function, and attempts to transform the function using it. For instance, if the transformation type can replace certain instructions within a function with functionally equivalent ones, a random subset of those instructions will be selected for replacement. The attempted transformation is applied only if the DNN becomes more likely to misclassify the binary.

Alg. 2 presents the pseudocode of the attack in the white-box setting. The algorithm starts by transforming all the functions in the binary in an undirected way. Namely, for each function in the binary, a transformation type is selected at random from the set of available transformations. The transformation is then applied to that function. When there are multiple ways to apply the transformation to the function, one is chosen at random. The algorithm then proceeds to further transform the binary for up to *niters* iterations. Each iteration starts by computing the embedding of the binary to a vector space, $\hat{e}$, and the gradient, g, of the DNN's loss function, $Loss_{\mathbb{F}}$, with respect to the embedding (lines 4–5). The loss function we use is the Carlini and Wagner loss function ($Loss_{cw}$) presented in Chap. 2. Ideally, to move the binary closer to misclassification, we would manipulate the binary so that the difference of its embedding from $\hat{e} + \alpha g$ (for some scaling factor α) is minimized (see prior work for examples [101, 108]). However, if applied without proper care, such manipulation would likely change the functionality of the binary or cause it to become ill-formed. Instead, we transform the binary via functionality-preserving transformations. As the transformation types are stochastic and may have many possible outputs (in some cases, more than can be feasibly enumerated), we cannot estimate their impact on the binary a priori. Therefore, we transform each function, f, by attempting to apply a (randomly picked) functionality-preserving transformation type at random (once per iteration); we apply the transformation only if it shifts the embedding in a direction similar to g (lines 6–14). More concretely, if g_f is the gradient with respect to the embedding of the bytes corresponding to f, and δ_f is the difference between the embedding of f's bytes after the attempted transformation and its bytes before, then the transformation is applied only if the cosine similarity (or, equivalently, the dot product) between g_f and δ_f is positive. Other optimization methods (e.g., genetic programming [227]) and similarity measures (e.g., similarity in the Euclidean space) that we tested did not perform as well.

If the input were continuous, it would be possible to perform the same attack in a black-box setting after estimating the gradients by querying the model (e.g., [85]). In our case, however, it is not possible to estimate the gradients of the loss with respect to the input, as the input is discrete. Therefore, the black-box attack we propose follows a general hill-climbing approach (e.g., [200]) rather than gradient ascent. The black-box attack is conceptually similar to the white-box one, and differs only in the condition checking whether to apply attempted transformations: Whereas the white-box attack uses gradient-related information to decide whether to apply a transformation, the black-box attack queries the model after attempting to transform a function, and accepts the transformation only if the probability of the target class increases.

Transformation Types We consider two families of transformation types [105, 158], as well as their combination. For the first family, we adopt and extend the transforma-

Algorithm 2: White-box attack against malware detection.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $Loss_{\mathbb{F}}$, x, y, $niters$
Output: $\hat{x}$

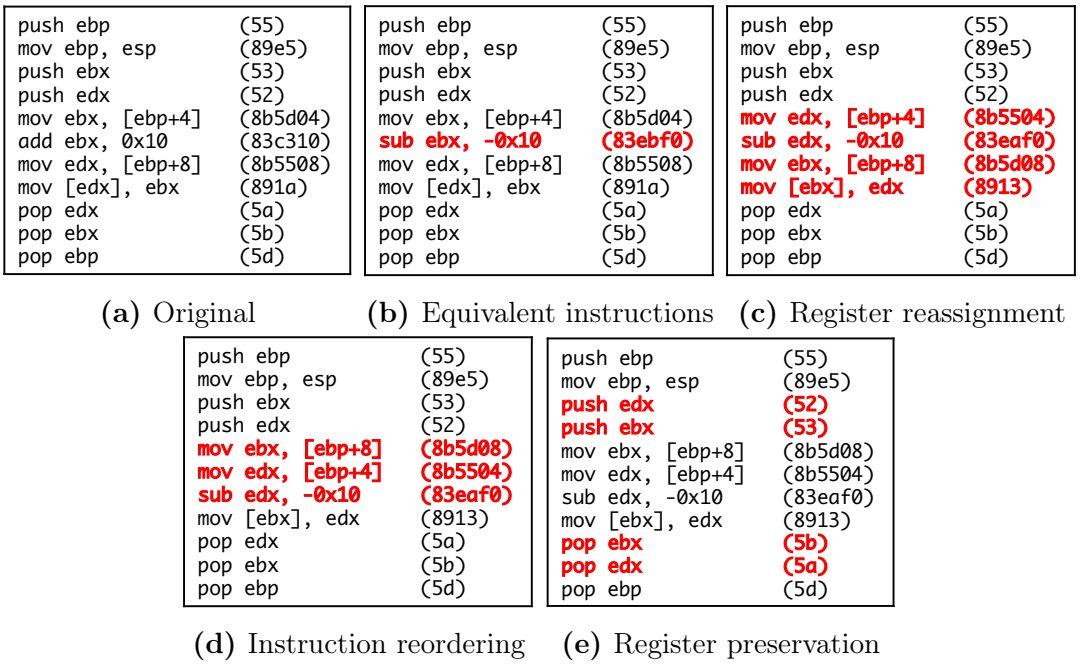
$i \leftarrow 0$;
$\hat{x} \leftarrow RandomizeAll(x)$;
while $\mathbb{F}(\hat{x}) = y$ *and* $i < niters$ **do**
  $\hat{e} \leftarrow \mathbb{E}(\hat{x})$;
  $g \leftarrow \frac{\partial Loss_{\mathbb{F}}(\hat{x},y)}{\partial \hat{e}}$;
  for $f \in \hat{x}$ **do**
    $o \leftarrow RandomTransformationType()$;
    $\tilde{x} \leftarrow RandomizeFunction(\hat{x}, f, o)$;
    $\tilde{e} \leftarrow \mathbb{E}(\tilde{x})$;
    $\delta_f = \tilde{e}_f - \hat{e}_f$;
    if $g_f \cdot \delta_f > 0$ **then**
      $\hat{x} \leftarrow \tilde{x}$;
  $i \leftarrow i + 1$;
return $\hat{x}$;

tion types proposed in the in-place randomization (*IPR*) work of Pappas et al. [158]. Given a binary to randomize, Pappas et al. proposed to disassemble it and identify functions and basic blocks, statically perform four types of transformations that preserve the functionality of the code, and then update the binary accordingly from the modified assembly code. The four transformation types considered are: *1)* to replace instructions with equivalent ones of the same length (e.g., `sub eax,4` → `add eax,-4`); *2)* to reassign registers within functions or a set of basic blocks (e.g., swap all instances of `ebx` and `ecx`) if this does not affect code that follows; *3)* to reorder instructions, using a dependence graph to ensure that no instruction appears before another one it depends on; and *4)* to change the order in which register values are pushed to and popped from the stack to save them across function calls.

To maintain the semantics of the code, the disassembly and transformations are performed conservatively (e.g., speculative disassembly, a disassembly technique that has a relatively high likelihood of misidentifying code, is avoided). *IPR* does not alter binaries' sizes and has no measurable effect on their run time [158].

The original implementation of Pappas et al. could not be used to evade malware detection due to several limitations, including ones preventing it from producing the majority of functionally equivalent binary variants that are conceptually achievable under the four transformation types. Thus, we extend and improve the implementation in various ways. First, we enable the transformations to compose. In other words, unlike Pappas et al.'s implementation, our implementation allows us to iteratively apply different transformation types to the same function. Second, we apply transformations more conservatively to ensure that the functionality of the binaries is preserved (e.g., by not replacing `add` and `sub` instructions if they are followed by instructions that read the flags register). Third, compared to the previous implementation, our implementation can handle a larger number of instructions and additional function-calling conventions. In particular, our implementation can rewrite binaries containing addi-

tional instructions (e.g., `shrd`, `shld`, `ccmove`) as well as less common calling conventions (e.g., nonstandard returns via increment of `esp` followed by a `jmp` instruction) without impacting the binaries' functionality. Last, we fix bugs in the original implementation (e.g., incorrect checks for writes to memory after reads). Fig. 4.1 shows an example of transforming code using *IPR*.



(a) Original **(b)** Equivalent instructions **(c)** Register reassignment

(d) Instruction reordering **(e)** Register preservation

Figure 4.1: An illustration of *IPR*. We show how the original code (a) changes after replacing instructions with equivalent ones (b), reassigning registers (c), reordering instructions (d), and changing the order of instructions that save register values (e). We provide the hex encoding of each instruction to its right. The affected instructions are boldfaced and colored in red.

The second family of transformation types that we build on is based on code displacement (*Disp*), proposed by Koo and Polychronakis [105]. Similarly to *IPR*, *Disp* begins by conservatively disassembling the binary. The original idea of *Disp* is to move code that can be leveraged as a gadget in code-reuse attacks to a new executable section in order to break the gadget. The original code to be displaced has to be at least five bytes in size so that it can be replaced with a `jmp` instruction that passes the control to the displaced code. If the displaced code contains more than five bytes, the bytes after the `jmp` are replaced with trap instructions that terminate the program; these would be executed if a code-reuse attack is attempted. In addition, another `jmp` instruction is appended immediately after the displaced code to pass the control back to the instruction that should follow. Of course, any displaced instruction that uses an address relative to the instruction-pointer (i.e., `IP`) register is also updated to reflect the new address after displacement. *Disp* has a minor effect on binaries' sizes (∼2% increase on average) and causes a small amount of run-time overhead (<1% on average) [105].

We extend *Disp* in two primary ways. First, we make it possible to displace any set of consecutive instructions within the same basic block, not only ones that belong to

$S \rightarrow Atom \mid S \cdot S \mid$
`bswp r` $\cdot S \cdot$ `bswp r` $\mid$
`xchg rh, rl` $\cdot S \cdot$ `xchg rh, rl` $\mid$
`push r` $\cdot S_r \cdot$ `pop r` $\mid$
`pushfd` $\cdot S_{ef} \cdot$ `popfd`
$Atom \rightarrow \Phi \mid$ `nop` $\mid$ `mov r, r`
$S_r \rightarrow S \mid S_r \cdot S_r \mid$ `pushfd` $\cdot S_{ef,r} \cdot$ `popfd`
$S_{ef} \rightarrow S \mid S_{ef} \cdot S_{ef} \mid$
`arth r, v` $\cdot S_{ef} \cdot$ `invarth r, v` $\mid$
`push r` $\cdot S_{ef,r} \cdot$ `pop r`
$S_{ef,r} \rightarrow S \mid S_r \mid S_{ef} \mid S_{ef,r} \cdot S_{ef,r} \mid$
`arth r, v` $\cdot S_{ef,r} \mid$
`logic r, v` $\cdot S_{ef,r}$

Figure 4.2: A context-free grammar for generating semantic nops. S is the starting symbol, Φ is the empty string, the symbol `arth` indicates an arithmetic operation (specifically, `add`, `sub`, `adc`, or `sbb`), `invarth` indicates its inverse, `logic` indicates a logical operation (specifically, `and`, `or`, or `xor`), and `r` and `v` indicate a register and a randomly chosen integer, respectively.

gadgets. Second, instead of replacing the original instructions with traps, we replace them with *semantic nops*—sets of instructions that *cumulatively* do not affect the memory or register values and have no side effects [39]. These semantic nops get jumped to immediately after the displaced code is done executing. To create the semantic nops, we use the context-free grammar described in Fig. 4.2. At a high-level, a semantic nop can be an atomic instruction (e.g., `nop`), or recursively defined as an invertible instruction that is followed by a semantic nop and then by the inverse instruction (e.g., `push eax` followed by a semantic nop and then by `pop eax`), or as two consecutive semantic nops. When the flags register's value is saved (i.e., between `pushfd` and `popfd` instructions), a semantic nop may contain instructions that affect flags (e.g., add and then subtract a value from a register), and when a register's value is saved too (i.e., between `push r` and `pop r`), a semantic nop may contain instructions that affect the register (e.g., decrement it by a random value). Using the grammar for generating semantic nops, for example, one may generate a semantic nop that stores the flags and `ebx` registers on the stack (`pushfd`; `push ebx`), performs an operation that might affect both registers (e.g., `add ebx, 0xff`), and then restores the registers (`pop ebx`; `popfd`).

When using *Disp*, our attacks start by displacing code up to a certain budget, to ensure that the resulting binary's size does not increase above a threshold (e.g., 1% above the original size). We first divide the budget (expressed as the number of bytes to be displaced) by the number of functions in the binary, and we attempt to displace exactly that number of bytes per function. If multiple options exist for what code in a function to displace, the code to be displaced is chosen at random. If a function does not contain enough code to displace, then we attach semantic nops (occupying the necessary number of bytes) after the displaced code to meet the per-function budget. In the rare case that the function does not have any basic block larger than five bytes, we

skip that function. Fig. 4.3 illustrates an example of displacement where semantic nops are inserted to replace original code, as well as after the displaced code, to consume the budget. Then, in each iteration of modifying the binary to cause it to be misclassified, new semantic nops are chosen at random and used to replace the previously inserted semantic nops if that moves the binary closer to misclassification.

```
...       ...              ...
0x4587:   add ax, 0x10     (6683c010)
0x458b:   sub bx, 0x10     (6683eb10)
0x458f:   cmp ax, bx       (6639d8)
...       ...              ...
```

(a) Original code

```
...       ...              ...
0x4587:   jmp 0x4800       (e974020000)
0x458c:   mov cx, cx       (6689c9)
0x458f:   cmp ax, bx       (6639d8)
...       ...              ...

...       ...              ...
0x4800:   add ax, 0x10     (6683c010)
0x4804:   sub bx, 0x10     (6683eb10)
0x4808:   nop              (90)
0x4805:   pushfd           (9c)
0x4806:   push ebx         (53)
0x4807:   add ebx, 0x1a    (83c31a)
0x480a:   pop ebx          (5b)
0x480b:   popfd            (9d)
0x480d:   jmp 0x458c       (e97afdffff)
...       ...              ...
```

(b) After *Disp*

Figure 4.3: An example of displacement. The two instructions staring at address `0x4587` in the original code (a) are displaced to to starting address `0x4800`. The original instructions are replaced with a `jmp` instruction and a semantic nop. To consume the displacement budget, semantic nops are added immediately after the displaced instructions and just before the `jmp` the passes the control back to the original code. Semantic nops are shown in boldface and red.

Some of the semantic nops contain integer values that can be set arbitrarily (e.g., see line 12 of Fig. 4.2). In a white-box setting, the bytes of the binary that correspond to these values can be set to perturb the embedding in the direction that is most similar to the gradient. Namely, if an integer value in the semantic nop corresponds to the i^{th} byte in the binary, we set this i^{th} byte to $b \in \{0, \ldots, 255\}$ such that the cosine similarity between $\mathbb{E}(b) - \mathbb{E}(\hat{x}_i)$ and g_i is maximized. This process is repeated each time a semantic nop is drawn to replace previous semantic nops in white-box attacks.

Prior work has suggested methods for detecting and removing semantic nops from binaries [40]. Such methods might appear viable for defending against *Disp*-based attacks, though as we discuss in Sec. 4.4, attackers can leverage various techniques to evade semantic-nop detection and removal.

Limitations While our implementation extends prior implementations, it can still be further improved. For instance, our implementation does not displace code that has been displaced in early iterations. A more comprehensive implementation might

apply displacements recursively. Furthermore, the composability of *IPR* and *Disp* transformations can be enhanced. Particularly, when applying both the *Disp* and *IPR* transformations to a binary, both types of transformations affect the original instructions of the binary. However, *IPR* does not affect the semantic nops that are introduced by *Disp*. Although there remains room for improvement, we did not pursue the remaining engineering endeavors because the attacks were successful despite the shortcomings of the implementation.

Furthermore, similarly to the original *Disp* and *IPR* implementations, there are no provable guarantees that our transformations preserve functionality. While our empirical experiments in Sec. 4.3.6 showed that the I/O behavior of the binaries we tested remained the same after the transformations, it may be possible to provably guarantee that functionality is preserved by using techniques from superoptimization (e.g., [181]), or by lifting binaries to an intermediate language where it would be more feasible to show functional equivalence between binaries (e.g., [25]).

4.3 Evaluation

In this section, we provide a comprehensive evaluation of our attack. We begin by providing details about the DNNs and data used for evaluation. We then show that naïve, random, transformations that are not guided via optimization do not lead to misclassification. Subsequently, we provide an evaluation of variants of our attack under a white-box setting, and compare with prior work. Then, we move to discuss evaluations of our attack in the black-box setting, both against the DNNs and commercial anti-viruses. We close the section with experiments to validate that the attacks preserve functionality.

4.3.1 Datasets and Malware-Detection DNNs

To train malware-detection DNNs, we used malicious binaries from a publicly available dataset that we augmented with benign binaries from standard software packages, as is standard (e.g., [102, 108]). In particular, we used malware binaries from nine malware families[1] that were published as part of a malware-classification competition organized by Microsoft [171]. This dataset contains raw binaries of malware samples targeting Windows machines. As such, the binaries adhere to the Portable Executable format (*PE*; the standard format for `.dll` and `.exe` files) [95]. However, to maintain sterility and prevent the binaries from executing, the curators removed their *PE* headers (which, among others, contain the entry points of the code). In total, the dataset contains 21,741 binaries that were partitioned into training and test sets by the dataset curators. We further partitioned the test set randomly into one group for validation (i.e., model and hyperparameter tuning) and another for final testing. Table 4.1 lists the number of binaries in the training, validation, and test sets.

Prior work [5, 107, 165] used larger malware datasets for training (in some cases containing two orders of magnitude more samples than the Microsoft dataset). Unfortunately, however, the raw binaries from prior work's datasets are proprietary. Conse-

[1] Gatak, Kelihos v1, Kelihos v3, Lollipop, Obfuscator ACY, Ramnit, Simda, Tracur, and Vundo.

Group	Train	Val.	Test
Benign	10,349	3,848	5,337
Malicious	10,868	5,257	5,616

Table 4.1: The number of benign and malicious binaries used to train, validate, and test the DNNs.

quently, we resorted to using a publicly available dataset. Nonetheless, the DNNs that we trained achieve comparable performance to those of prior work.

To collect benign binaries, we installed standard packages on a newly created 32-bit Windows 7 virtual machine and gathered the *PE* binaries pertaining to these packages. Specifically, we used the Ninite and Chocolatey[2] package managers to install 179 packages. The packages that we installed included popular ones that are commonly used by a variety of users (such as Chrome, Firefox, WinRAR, Spotify, ...), as well as packages that are likely to be used by specific user groups, such as developers (e.g., PyCharm), academics (e.g., MiKTeX), and graphics designers (e.g., Gimp). This resulted in 19,534 binaries that we partitioned into training, validation, and test sets of comparable sizes to those for malware (see Table 4.1). When partitioning, we placed binaries from the same packages in the same partitions to ensure that the DNNs learned to tell apart malicious and benign binaries rather to than to associate binaries of the same packages with each other.

Using the malicious and benign samples, we trained two malware-detection DNNs. Both DNNs receive binaries' raw bytes as inputs and output the probability that the binaries are malicious. The first DNN, proposed by Krčál et al. [107], receives inputs up to 512 KB in size. We refer to it by *AvastNet*, in reference to the authors' affiliation. The second DNN, proposed by Raff et al. [165], receives inputs up to 2 MB in size. We refer to this DNN by *MalConv*, as per the authors' naming. Except for the batch-size parameter, we used the same training parameters reported in the papers. We set the batch size to 32 due to memory limitations. In addition, when using benign binaries for training, we excluded the headers. This is both to remain consistent with the malicious binaries (which do not include headers), but also to ensure that the DNNs would not rely on header values that are easily manipulable for classification [49]. As the results below demonstrate, excluding the header leads to DNNs that are more difficult to evade.

The classification performance of the DNNs is reported in Table 4.2. Both DNNs achieve test accuracy of about 99%. Even when restricting the false positive rates (FPRs) conservatively to 0.1% (as is often done by anti-virus vendors [107]), the true positive rates (TPRs) remain as high as 80–89% (i.e., 80–89% of malicious binaries are detected). The performance results that we computed are superior to the ones reported in the original papers both for classification from raw bytes and from manually crafted features [107, 165]. We believe the reason to be that our dataset was restricted to nine

[2] `https://ninite.com/` and `https://chocolatey.org/`

DNN	Accuracy Train	Val.	Test	TPR @ 0.1% FPR
AvastNet	99.23%	98.29%	98.92%	80.28%
MalConv	99.96%	98.33%	99.15%	88.73%

Table 4.2: The DNNs' performance. We report the accuracies on the different data partitions, as well as the TPR at the operating point where the FPR equals 0.1%.

malware families, and expect the performance to slightly decrease when incorporating additional malware families.

In addition to the two DNNs that we trained, we evaluated the attacks using a publicly available DNN that was trained by Anderson and Roth [5]. We refer to this DNN by *Endgame*, in reference to the authors' affiliation. *Endgame* has a similar architecture to *MalConv*. The salient differences are that: *1) Endgame*'s input dimensionality is 1 MB (compared to 2 MB for *MalConv*); and *2) Endgame* uses the *PE* header for classification. On a dataset separate from ours that was created by a computer-security company, *Endgame* achieved about 92% TPR when the FPR was restricted to 0.1% [5].

To evaluate attacks against the DNNs, we selected binaries according to three criteria. First, the binaries had to be unpacked. To this end, we used standard packer detectors (specifically, Packerid [184] and Yara [214]) and deemed binaries as unpacked only if none of the detectors exhibited a positive detection. This method is similar to the one followed by Biondi et al. [20].[3] While the data used to train and evaluate the performance of the DNNs included packed binaries (we could not exclude potentially packed binaries from the Microsoft dataset due to missing headers), the high accuracy of the DNNs on the test samples suggests that the DNNs' performance was not impacted by (lack of) packing. Second, the binaries had to be classified correctly and with high confidence by the DNNs that we trained. In particular, malicious (resp., benign) binaries had to be classified as malicious (resp., benign), and the estimated probability that they are malicious had to be above (resp., below) the threshold where the FPR (resp., false negative rate, FNR) is 0.1%. Consequently, our evaluation of the attacks' success is conservative: the attacks would be more successful for binaries that are initially classified correctly, but not with high confidence. Third, the binaries' sizes had to be smaller than the DNNs' input dimensionality. While the DNNs can classify binaries whose size is larger than the input dimensionality (as can be seen from the high classification accuracy on the validation and test sets), we avoided large binaries as a means to prevent evasion by displacing malicious code outside the input range of the DNNs.

Using these criteria, we selected 99 benign binaries from the test set to evaluate the attacks against each of the three DNNs. Leading malware detection to misclassify these benign samples can harm users' trust in the defense [76]. Unfortunately, we were unable to use the malicious binaries from the Microsoft dataset to evaluate our attacks, as they lack the *PE* headers, and so they cannot be disassembled as necessary

[3]Biondi et al. used three packer-detection tools instead of two. Unfortunately, we were unable to get access to one of the proprietary tools.

Group	*AvastNet*	*MalConv*	*Endgame*
Benign	99	99	99
Malicious	72	95	86

Table 4.3: The number of benign and malicious binaries used to test our attacks against the three DNNs.

for the *IPR* and *Disp* transformations. To this end, we used VirusShare [169]—an online repository of malware samples—to collect malicious binaries belonging to the nine families that are present in the Microsoft dataset (as indicated by the labels that commercial anti-viruses assigned the samples). Following this approach, we collected a variable number of binaries that were unseen in training to test the attacks against each one of the DNNs, as specified in Table 4.3. The total number of samples the we collected to evaluate the attacks is comparable to that used in prior work on evading malware detection [101, 108, 200, 205].

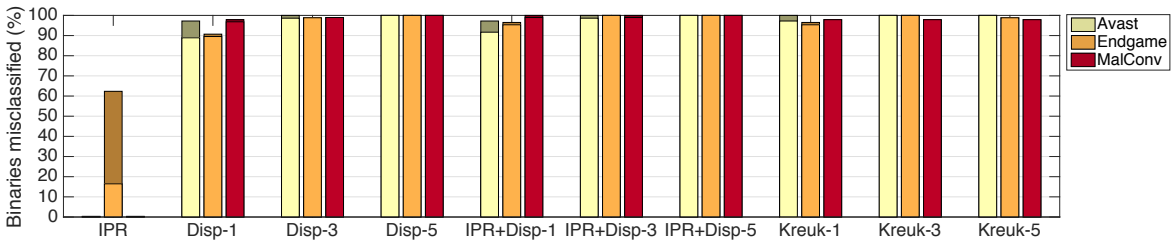
4.3.2 Randomly Applied Transformations

We first evaluated whether naïvely transforming binaries at random would lead to evading the DNNs. To do so, for each binary that we used to evaluate the attacks we created 200 variants using the *IPR* and *Disp* transformations and classified them using the DNNs. If any of the variants was misclassified by a DNN, we would consider the evasion attempt successful. We set *Disp* to increase binaries' sizes by 5% (i.e., the displacement budget was set to 5% of the binary's original size). We selected 200 and 5% as parameters for this experiment because our attacks were executed for 200 iterations at most, and achieved almost perfect success when increasing binaries' sizes by 5% (see below).

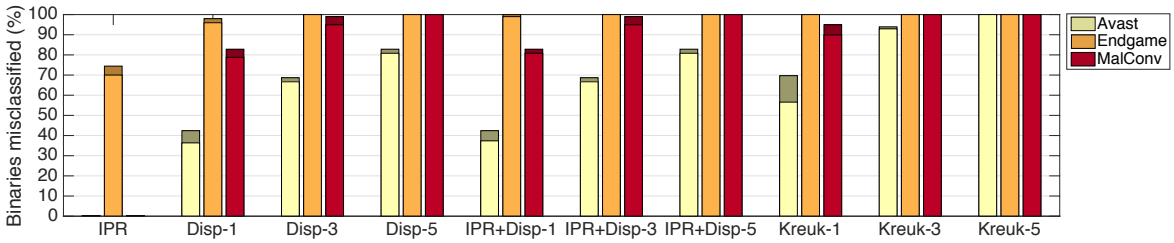
Except for a single benign binary that was misclassified by *MalConv* after being transformed, no other misclassification occurred. Hence, we can conclude that the DNNs are robust against naïve transformations, and that more principled approaches are needed to mislead them.

4.3.3 White-box Attacks vs. DNNs

In the white-box setting, we evaluated seven variants of our attack. One variant, to which refer by *IPR*, relies on the *IPR* transformations. Three variants, *Disp*-1, *Disp*-3, and *Disp*-5, rely on the *Disp* transformations, where the numbers indicate the displacement budget as a percentage of the binaries' sizes (e.g., *Disp*-1 increases binaries' sizes by 1%). The last three attack variants, *IPR*+*Disp*-1, *IPR*+*Disp*-3, and *IPR*+*Disp*-5, use the *IPR* and *Disp* transformations combined. We executed the attacks up to 200 iterations and stopped early if the binaries were misclassified with high confidence. For malicious (resp., benign) binaries, this meant that they were misclassified as benign (resp., malicious) with an estimated probability that they are malicious below the probability where the FPR (resp., FNR) is 0.1%. We set 5% as the



(a) Malicious binaries



(b) Benign binaries

Figure 4.4: Attacks' success rates in the white-box setting. For each attack and DNN, we provide the percentage of misclassified malicious (a) and benign (b) binaries. The brightly colored bars show the percentage of binaries that were misclassified with high confidence.

maximum displacement budget and 200 as the maximum number of iterations, as we empirically found that the attacks were almost always successful with these parameters.

In addition to our attacks, we implemented and evaluated an attack proposed by Kreuk et al. [108]. To mislead DNNs, the attack of Kreuk et al. appends adversarially crafted bytes to binaries. These bytes are crafted via an iterative algorithm that first computes the gradient g_i of the loss with respect to the embedding $\mathbb{E}(x_i)$ of the binary x_i at the i^{th} iteration, and then sets the adversarial bytes to minimize the L_2 distance of the new embedding $\mathbb{E}(x_{i+1})$ from $\mathbb{E}(x_i)+\epsilon sign(g_i)$, where ϵ is a scaling parameter. We tested three variants of the attack, denoted by *Kreuk*-1, *Kreuk*-3, and *Kreuk*-5, which increase the binaries' sizes by 1%, 3%, and 5%, respectively. As a loss function, we used $Loss_{cw}$. Similarly to our attacks, we executed Kreuk et al.'s attacks up to 200 iterations, stopping sooner if misclassification with high confidence occurred. Furthermore, we set ϵ=1, as we empirically found that it leads to high evasion success.

We measured the success rates of attacks by the percentage of binaries that were misclassified. The results of the experiment are provided in Fig. 4.4. One can immediately see that attacks using the *Disp* transformations were more successful than *IPR*. In fact, *IPR* was able to mislead only *Endgame*, achieving 62% success rate at misclassifying malicious binaries as benign, and 74% success rate at misclassifying benign binaries as malicious. *IPR* was unable to mislead *AvastNet* and *MalConv* in any attempt. This indicates that using binaries' headers for classification, similarly to *Endgame*, may lead to more vulnerable models than when excluding the header.

In contrast to *IPR*, other variants of our attack achieved considerable success. For example, *Disp*-5 achieved high-confidence misclassification in all attempts, except when attempting to mislead *AvastNet* to misclassify benign binaries, where 81% of the attempts succeeded. As one would expect, attacks with higher displacement budget

were more successful. Specifically, attacks with 5% displacement budget were more successful than ones with 3%, and the latter were more successful than attacks with 1% displacement budget.

In addition to achieving higher success rates, another advantage of *Disp*-based attacks over *IPR*-based ones is their time efficiency. While displacing instruction at random from within a function with n instructions has $\mathcal{O}(n)$ time complexity, certain *IPR* transformations have $\mathcal{O}(n^2)$ time complexity. For example, reordering instructions requires building a dependence graph and extracting instructions one after the other. If every instruction in a function depends on previous ones, this process takes $\mathcal{O}(n^2)$ time. In practice, we found that while *Disp*-based attacks took about 159 seconds to run on average, *IPR*-based ones took 606 seconds.

While *IPR* had limited success, combining *IPR* with *Disp* achieved higher success rates than respective *Disp*-only attacks with the same displacement budget. For example, *IPR*+*Disp*-1 had 6% higher success rate than *Disp*-1 when misleading *Endgame* to misclassify a malicious binary as benign (97% vs. 91% success rate). Thus, in certain situations, *Disp* and *IPR* can be combined to fool the DNNs while increasing binaries' sizes less than *Disp* alone.

The variants of Kreuk et al.'s attack achieved success rates comparable to variants of our attack. For example, *Kreuk*-5 was almost always able to mislead the DNNs—it achieved 99% and 98% success rate when attempting to mislead *Endgame* and *MalConv*, respectively, to misclassify malicious binaries, and 100% success rate in all the other attempts. One can also see that the success rate increased as the attacks increased the binaries' sizes. In particular, *Kreuk*-5 was more successful at misleading the DNNs than *Kreuk*-3, which, in turn, was more successful than *Kreuk*-1.

While Kreuk et al.'s attack achieved success rates that are comparable to ours, it is important to highlight that their attack is easier to defend against. As a proof of concept, we implemented a sanitization method to defend against the attack. The method finds all the sections that do not contain instructions (using the IDAPro disassembler [77]) and masks the sections' content with zeros. As Kreuk et al.'s attack does not introduce code to the binaries, the defense masks the adversarial bytes that it introduces. Consequently, the evasion success rates of the attack drop significantly. For example, the success rates of *Kreuk*-5 against *MalConv* drop to 1% and 11% for malicious and benign binaries, respectively. At the same time, the defense has little-to-no effect on our attacks. For example, *Disp*-5 achieves 91% and 100% success rates for malicious and benign binaries, respectively. Moreover, the classification accuracy remains high for malicious (99%) and benign (93%) binaries after the defense.

4.3.4 Black-box Attacks vs. DNNs

As explained in Sec. 4.2, because the DNNs' input is discrete, estimating gradient information to mislead them in a black-box setting is not possible. To this end, the black-box version of Alg. 2 uses a hill-climbing approach to query the DNN after each attempted transformation to decide whether to keep the transformation. Because querying the DNNs after each attempted transformation leads to a significant increase in run time of the attacks ($\sim$30$\times$ on a machine with GeForce GTX 980 GPU), we limited our experiments to *Disp* transformations with a displacement budget of 5%, and attempted

to mislead the DNNs to misclassify malicious binaries. We executed the attacks up to 200 iterations and stopped early if misclassification occurred.

The attacks were most successful against *Endgame*, achieving a success rate of 97%. In contrast, 33% of the evasion attempts against *MalConv* succeeded, and none of attempts against *AvastNet* were successful. In cases of failure, we observed that the optimization process was getting stuck in local minimas. We believe that it may be possible to enhance the performance of the attacks in such cases by deploying methods for overcoming local minimas (e.g., the Metropolis algorithm [151], or Monte-Carlo tree search [194]).

Motivated by the universality property of adversarial examples [144] and the finding of Kreuk et al. that adversarial bytes generated for one binary may lead to evasion when appended to another binary [108], we wanted to see if the transformations applied to one binary can lead to evasion when applied to another. If so, attackers in a black-box setting may invest considerable effort to transform one binary to mislead malware-detection and then apply the same transformations to other binaries. We focused on *Disp* transformations and tested whether the semantic nops that were added to certain binaries via the *Disp*-5 attack in the white-box setting lead to evasion when added to other binaries. To do so, we developed a modified version of *Disp* that displaces code within binaries at random, but instead of drawing the semantic nops randomly, it borrows them from another previously transformed binary.

We tested this approach with malicious binaries and found that it leads to relatively high success rates. For *AvastNet* and *Endgame*, certain transformations led to evasion success rates as high as 75% and 86%, respectively, when borrowed from one binary and applied to other binaries (i.e., merely 14%–25% lower success rates than for white-box attacks). The success rates were more limited against *MalConv*, achieving a maximum of 24%.

4.3.5 Transferability of Attacks to Commercial Anti-Viruses

To assess whether our attacks affect commercial anti-viruses, we tested how the binaries that were misclassified by the DNNs with high confidence in the white-box setting get classified by anti-viruses available via VirusTotal [41]—an online service that aggregates the results of 68 commercial anti-viruses. Since anti-viruses often rely on ML for malware detection, and since prior work has shown that adversarial examples that evade one ML model often evade other models (a phenomenon called transferability) [66, 156], we expected that the malicious (resp., benign) binaries generated by our attacks would be classified as malicious by fewer (resp., more) anti-viruses than the original binaries.

As a baseline, we first classified the original binaries using the VirusTotal anti-viruses. As one would expect, all the malicious binaries were detected by several anti-viruses. The median number of anti-viruses that detected any particular malware binary as malicious was 55, out of 68 total anti-viruses. In contrast, the original benign binaries were detected by a median of 0 anti-viruses, with a total of five false positives across all binaries and anti-viruses. To further gauge the accuracy of the commercial anti-viruses, we used them to classify binaries that were transformed at random using the *Disp* and *IPR* transformation types (in the same manner as Sec. 4.3.2). We found that certain anti-viruses were susceptible to such simple evasion attempts—the

DNN	*IPR*	*Disp*-1	*Disp*-3	*Disp*-5	*IPR*+*Disp*-1	*IPR*+*Disp*-3	*IPR*+*Disp*-5
AvastNet	-	**36**	**35**	**36**	**36**	**35**	**36**
Endgame	**33**	**35**	**36**	**35**	**35**	**36**	**35**
MalConv	-	**36**	**35**	**36**	**36**	**35**	**36**

(a) Malicious binaries

DNN	*IPR*	*Disp*-1	*Disp*-3	*Disp*-5	*IPR*+*Disp*-1	*IPR*+*Disp*-3	*IPR*+*Disp*-5
AvastNet	-	**2**	**2**	**1**	**2**	**2**	**1**
Endgame	1	**2**	**2**	**2**	**2**	**2**	**2**
MalConv	-	**1**	**1**	**1**	**1**	**1**	**1**

(b) Benign binaries

Table 4.4: The median number of VirusTotal anti-viruses that positively detected (i.e., as malicious) malicious (a) and benign (b) binaries that were transformed by our white-box attacks (columns) to mislead the different DNNs (rows). The median number of anti-viruses that positively detected for the original malicious and benign binaries is 55 and 0, respectively. Cases in which the change in the number of detections is statistically significant are in bold.

median number of anti-viruses that detected the malicious binaries correctly decreased to 42. At the same time, the median number of anti-viruses that detected benign binaries as malicious remained 0. Presumably, some anti-viruses were evaded by random transformations due to using fragile detection mechanisms, such as signatures.

Table 4.4 summarizes the effect of our attacks on the number of positive detections (i.e., classification of binaries as malicious) by the anti-viruses. Compared to the original malicious binaries and ones that were transformed at random, the malicious binaries transformed by our attacks were detected as malicious by fewer anti-viruses. The median number of anti-viruses that correctly detected the malicious binaries decreased from 55 for the original binaries and 42 for ones transformed at random to 33–36, depending on the attack variant and the targeted DNN. According to a Kruskal-Wallis test, this reduction is statistically significant ($p < 0.01$ after Bonferroni correction). In other words, the malicious binaries that were transformed by our attacks were detected by only 49%–53% of the VirusTotal anti-viruses in the median case.

The number of positive detections of benign binaries increased after they were transformed by our attacks: The median number of anti-viruses that detected the benign binaries as malicious was one or two, depending on the attack variant and the targeted DNN. In certain cases, the number of positive detections was as high as 19 (i.e., 28% of the VirusTotal anti-viruses reported the binary as malicious). Except for one attack (*IPR* targeting *Endgame*), the increase in the number of detections is statistically significant ($p < 0.01$ after Bonferroni correction), according to a Kruskal-Wallis test.

Our attacks evaded a larger number of anti-viruses compared to random transformations, likely due to transferring from our DNNs to ML detectors that are used by the anti-viruses. A glance at the websites of the anti-viruses' vendors showed that

15 of the 68 vendors explicitly advertise relying on ML for malware detection. These anti-viruses were especially susceptible to evasion by our attacks. Of particular note, one vendor advertises that it relies solely on ML for malware detection. This vendor's anti-virus misclassified 78% of the benign binaries that were produced by one variant of our attack as malicious. In general, a median of 1–2 anti-viruses (from the 15 vendors) misclassified benign binaries that were processed by our attacks as malicious. Even more concerning, a popular anti-virus whose vendor reports to rely on ML misclassified 85% of the malicious binaries produced by a variant of our attack as benign. Generally, malicious binaries that were produced by our attacks were detected by a median number of 7–9 anti-viruses of the 15—down from 12 positive detections for the original binaries. All in all, while online advertising (or lack thereof) is a weak indicator of the nature of detectors used by anti-viruses (e.g., some prominent vendors do not explicitly advertise the use of ML), our results support that binaries that were produced by our attacks were able to evade ML-based detectors that are used by anti-virus vendors.

4.3.6 Correctness

A key feature of our attacks is that they transform binaries to mislead DNNs while preserving their functionality. We followed standard practices from the binary-diversification literature [104, 105, 158] to ensure that the functionality of the binaries was kept intact after being processed by our attacks. First, we transformed ten different benign binaries (e.g., `python.exe` of Python version 2.7, and Cygwin's[4] `less.exe` and `grep.exe`) with our attacks and manually validated that they functioned properly after being transformed. For example, we were still able to search files with `grep` after the transformations. Second, we transformed the `.exe` and `.dll` files of a stress-testing tool[5] with our attacks and checked that the tool's tests passed after the transformations. Using stress-testing tools to evaluate the correctness of binary-transformation methods is common, as such tools are expected to cover most branches affected by the transformations. Third, and last, we also transformed ten malware binaries and used the Cuckoo Sandbox [69]—a popular sandbox for malware analysis—to check that their behavior remained the same. All ten binaries attempted to access the same hosts, IP addresses, files, APIs, and registry keys before and after being transformed.

4.4 Discussion

Our proposed attacks achieved high success rates at fooling DNNs for malware detection in white-box and black-box settings. The attacks were also able to mislead commercial anti-viruses, especially ones that leverage ML algorithms. To protect users and their systems, it is important to develop mitigation measures to make malware detection robust against evasion by our attacks. We now discuss potential mitigations to our attacks, followed by takeaways.

[4] `https://www.cygwin.com/`

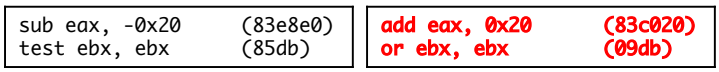
[5] `https://www.passmark.com/products/performancetest/`

4.4.1 Potential Mitigations

Prior Defenses We considered several prior defenses to mitigate our attacks, but, unfortunately, most showed little promise. For instance, adversarial training (e.g., [66, 113]) is infeasible, as the attacks are computationally expensive. Depending on the attack variant, it took an average of 159 or 606 seconds to run an attack. As a result, running just a single epoch of adversarial training would to take several weeks (using our hardware configuration), as each iteration of training requires running an attack for every sample in the training batch. Moreover, while adversarial training might increase the DNNs' robustness against attackers using certain transformation types, attackers using new transformation types may still succeed at evasion [55]. Defenses that provide formal guarantees (e.g., [103, 141]) are even more computationally expensive than adversarial training. Moreover, those defenses are restricted to adversarial perturbations that, unlike the ones produced by our attacks, have small L_∞- and L_2-norms. Prior defenses that transform the input before classification (e.g., via quantization [226]) are designed mainly for images and do not directly apply to binaries. Lastly, signature-based malware detection would not be effective, as our attacks are stochastic and produce different variants of the binaries after different executions.

Differently from prior attacks on DNNs for malware detection [101, 108, 205], our attacks do not merely append adversarially crafted bytes to binaries, or insert them between sections. Such attacks may be defended against by detecting and sanitizing the inserted bytes via static analysis methods (e.g., similarly to the proof of concept shown in Sec. 4.3.3, or using other methods [110]). Instead, our attacks transform binaries' original code, and extend binaries only by inserting instructions that are executed at run time at various parts of the binaries. As a result, our attacks are difficult to defend against via static or dynamic analyses methods (e.g., by detecting and removing unreachable code), especially when augmented by measures to evade these methods.

Binary normalization [6, 40, 216] is an approach that was proposed to enhance malware detection that seemed viable for defending against our attacks. The high-level idea of normalization is to employ certain transformations to map binaries to a standard form and thus undo attackers' evasion attempts before classifying the binaries as malicious or benign. For example, Christodorescu et al. proposed a method to detect and remove semantic nops from binaries before classification, and showed that it improves the performance of commercial anti-viruses [40]. To mitigate our *Disp*-based attacks, we considered using the semantic nop detection and removal method followed by a method to restore the displaced code to its original location. Unfortunately, we realized that such a defense can be undermined using *opaque predicates* [44, 146]. Opaque predicates are predicates whose value (w.l.g., assume true) is known a priori to the attacker, but is hard for the defender to deduce. Often, they are based on *NP*-hard problems [146]. Using opaque predicates, attackers can produce semantic nops that include instructions that affect the memory and registers only if an opaque predicate evaluates to false. Since opaque predicates are hard for defenders to deduce, the defenders are likely to have to assume that the semantic nops impact the behavior of the program. As a result, the semantic nops would survive the defenders' detection and removal attempts. As an alternative to opaque predicates, attackers can also use *evasive predicates*—predicates that evaluate to true or false with an overwhelming



(a) **(b)**

Figure 4.5: An example of normalizing code via *Eqv*. The original code (a) is transformed via *Eqv* (b) to decrease the lexicographic order.

probability (e.g., checking if a randomly drawn 32-bit integer is equal to 0) [16]. In this case, the binary will function properly the majority of the time, and may function differently or crash once every many executions.

The normalization methods proposed by prior work would not apply to the transformations performed by our *IPR*-based attacks. Therefore, we explored methods to normalize binaries to a standard form to undo the effects of *IPR* before classification. We found that a normalization process that leverages the *IPR* transformations to map binaries to the form with the lowest lexicographic representation (where the alphabet contains all possible 256 byte values) is effective at preventing *IPR*-based attacks. Formally, if $[x]$ is the equivalence class of binaries that are functionally equivalent to x and that can be produced via the *IPR* transformation types, then the normalization process produces an output $norm(x) \in [x]$, such that, $norm(x) \leq x_i$ for every $x_i \in [x]$. For each transformation type, we devise an operation that would decrease a binary's lexicographic representation when applied: *1)* instructions would be replaced with equivalent ones only if the new instructions are lexicographically lower (*Eqv*); *2)* registers in functions would be reassigned only if the byte representation of the first impacted instruction would decrease (*Regs*); *3)* instructions would be reordered such that each time we would extract the instruction from the dependence graph with the lowest byte representation that does not depend on any of the remaining instructions in the graph (*Ord1*); and *4)* `push` and `pop` instructions that save register values across function calls would be reordered to decrease the lexicographic representation while maintaining the last-in-first-out order (*Ord2*). Fig. 4.5 depicts an example of replacing one instruction with an equivalent one via *Eqv* to decrease the lexicographic order of code.

Unfortunately, as shown in Fig. 4.6, when the different types of transformation types are composed, applying individual normalization operations does not necessarily lead to the binary's variant with the minimal lexicographic representation, as the procedure may be stuck in a local minima. To this end, we propose a stochastic algorithm that is guaranteed to converge to binaries' normalized variants if executed for a sufficiently large number of iterations.

Alg. 3 presents a psuedocode of the normalization algorithm. The algorithm receives a binary x and the number of iterations *niters* as inputs. It begins by drawing a random variant of x, by applying all the transformation types to each function at random (line 1). The algorithm then proceeds to apply each of the individual normalization operations to decrease the lexicographic representation of the binary (lines 14–24), while self-supervising the normalization process. Specifically, the algorithm keeps track of the last iteration an operation decreased the binary's representation (line 19). If none of the four operations affect any of the functions, we deduce that the

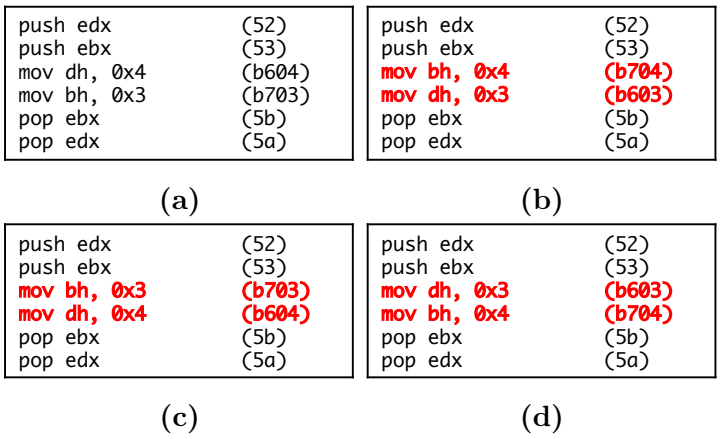


Figure 4.6: The normalization process can get stuck in a local minima. The lexicographic order of the original code (a) increases when reassigning registers (b) or reordering instructions (c). However, composing the two transformation (d) decreases the lexicographic order.

normalization process is stuck in a (global or local) minima, and a random binary is drawn again by randomizing all functions (limes 10–12), and the normalization process restarts.

When $niters \to \infty$ (i.e., the number of iterations is large enough), Alg. 3 would eventually converge to a global minima. Namely, it would find the variant of x with the minimal lexicographic representation. In fact, we are guaranteed to find $norm(x)$ even if we simply apply the transformation types at random x for $niters \to \infty$ iterations. When testing the algorithm with two binaries of moderate size, we found that $niters$=2,000 was sufficient to converge for the same respective variants after every run. These variants are likely to be the global minimas. However, executing the algorithm for 2,000 iterations is computationally expensive, and impractical within the context of a widely deployed malware-detection system. Hence, for the purpose of our experiments, we set $niters$=10, which we found to be sufficient to successfully mitigate the majority of attacks.

We executed the normalization algorithm using the malicious and benign binaries produced by the *IPR*-based attacks to fool *Endgame* in the white-box setting, and found that the success rates dropped to 3% and 0%, respectively, compared to 62% and 74% before normalization. At the same time, the classification accuracy over the original binaries was not affected by normalization. As our experiments in Sec. 4.3 have shown, generating functionally equivalent variants of binaries via random transformations results in correct classifications almost all of the time. Normalization of binaries to the minimal lexicographic representation deterministically leads to the specific functionally equivalent variants that get correctly classified with high likelihood.

Instruction Masking While normalization was useful for defending against *IPR*-based attacks, it cannot mitigate the more pernicious *Disp*-based attacks that are augmented with opaque or evasive predicates. Moreover, normalization has the general limitations that attackers could use transformations that the normalization algorithm is not aware of or could obfuscate code to inhibit normalization. Therefore, we explored

Algorithm 3: In-place normalization.

Input : x, $niters$
Output: x_{min}

$\hat{x} \leftarrow RandomizeAll(x)$;
$i, last_{update} \leftarrow 0, 0$;
$x_{min} \leftarrow \hat{x}$;
while $i < niters$ **do**
  if $i\%4 = 0$ **then**
    `// Normalization operations`
    $ops \leftarrow \{\}$;
    for $f \in \hat{x}$ **do**
      $ops[f] = \{Eqv, Regs, Ord1, Ord2\}$;
    `// If stuck, randomize and restart`
    if $i - last_{update} \geq 4$ **then**
      $\hat{x} \leftarrow RandomizeAll(x)$;
  for $f \in \hat{x}$ **do**
    $o \leftarrow pop(ops[f])$;
    $\tilde{x} \leftarrow Minimize(\hat{x}, f, o)$;
    if $\hat{x} \neq \tilde{x}$ **then**
      $\hat{x} \leftarrow \tilde{x}$;
      $last_{update} \leftarrow i$;
      if $\hat{x} < x_{min}$ **then**
        $x_{min} \leftarrow \hat{x}$;
  $i \leftarrow i + 1$;
return x_{min};

additional defensive measures. In particular, motivated by the fact that randomizing binaries without the guidance of an optimization process is unlikely to lead to misclassification, we explored whether masking instructions at random can mitigate attacks while maintaining high performance on the original binaries. The defense works by selecting a random subset of the bytes that pertain to instructions and masking them with zeros (a commonly used value to pad sections in binaries). While the masking is likely to result in an ill-formed binary that is unlikely to execute properly (if at all), the masking only occurs before classification, which does not require a functional binary. Depending on the classification result, one can decide whether or not to execute the unmasked binary.

We tested the defense on binaries generated via the *IPR+Disp*-5 white-box attack and found that it is effective at mitigating attacks. For example, when masking 25% of the bytes pertaining to instructions, the success rates of the attack decreases from 83%–100% for malicious and benign binaries against the three DNNs to 0%–20%, while the accuracy on the original samples was only slightly affected (e.g., it became 94% for *Endgame*). Masking less than 25% of the instructions' bytes was not as effective at mitigating attacks, while masking more than 25% led to a significant decrease in accuracy on the original samples.

Detecting Adversarial Examples To prevent binaries transformed with our attacks (i.e., adversarial examples) from fooling malware detection, defenders may attempt to deploy methods to detect them. In cases of positive detections of adversarial examples, defenders may immediately classify them as malicious (regardless of whether they were originally malicious or benign). For example, because *Disp*-based attacks increase binaries' sizes and introduce additional `jmp` instructions, defenders may train statistical ML models that use features such as binaries' sizes and the ratio between `jmp` instructions and other instructions to detect adversarial examples. While training relatively accurate detection models may be feasible, we expect this task to be difficult, as the attacks increase binaries' sizes only slightly (1%–5%), and do not introduce many `jmp` instructions (7% median increase for binaries transformed via *Disp*-5). Furthermore, approaches for detecting adversarial examples are likely to be susceptible to evasion attacks (e.g., by introducing instructions after opaque predicates to decrease the ratio between `jmp` instructions and others). Last, another risk that defenders should take into account is that the defense should be able to precisely distinguish between adversarial examples and non-adversarial benign binaries that are transformed by similar methods to mitigate code-reuse attacks [105, 158].

4.4.2 Takeaways

While masking a subset of the bytes that pertain to instructions led to better performance on adversarial examples, it was still unable to prevent all evasion attempts. Although the defense may raise the bar to attackers, and make attacks even more difficult if combined with a method to detect adversarial examples, attackers may be able to adapt to undermine them. For example, attackers may build on techniques for optimization over expectations to generate binaries that would mislead the DNNs even when masking a large number of instructions, in a similar manner to how attackers can evade image-classification DNNs under varying lighting conditions and camera angles [11, 57, 191, 192]. In fact, prior work has already demonstrated how defenses are often vulnerable to adaptive, more sophisticated, attacks [10]. Thus, since there is no clear defense to prevent attacks against the DNNs that we studied in this chapter, or even general methods to prevent attackers from fooling ML models via arbitrary perturbations, we advocate for augmenting malware-detection systems with methods that are not based on ML (e.g., ones using templates to reason about the semantics of programs [39]), and against the use of ML-only detection methods, as has become recently popular [47].

4.5 Conclusion

The work presented in the chapter proposes evasion attacks on DNNs for malware detection. Differently from prior work, the attacks do not merely insert adversarially crafted bytes to mislead detection. Instead, guided by optimization processes, our attacks transform the instructions of binaries to fool malware detection while keeping functionality of the binaries intact. As a result, these attacks are challenging to defend against. We conservatively evaluated different variants of our attack against three

DNNs under white-box and black-box settings, and found the attacks successful as often as 100% of the time. Moreover, we found that the attacks pose a security risk to commercial anti-viruses, particularly ones using ML, achieving evasion success rates of up to 85%. We explored several potential defenses, and found some to be promising. Nevertheless, adaptive adversaries remain a risk, and we recommend the deployment of multiple detection algorithms, including ones not based on ML, to raise the bar against such adversaries.

Chapter 5

A General Framework for Attacks with Objectives

5.1 Introduction

In the previous chapters (Chaps. 3–4), to create adversarial examples with multiple objectives we modeled the objectives in an ad hoc fashion. In contrast, in this chapter we propose a general framework for capturing multiple objectives in the process of generating adversarial examples. Our framework builds on Generative Adversarial Networks (GANs, see Chap. 5.2.3) to train an attack *generator*, i.e., a neural network that can generate attack instances (i.e., adversarial inputs) that meet certain objectives. Due to our framework's basis in GANs, we refer to it using the anagram AGNs, for *adversarial generative networks*.

To illustrate the utility of AGNs, we return to the task of printing eyeglasses to fool face-recognition systems [191] and demonstrate how to accommodate a number of types of objectives within it. Specifically, we use AGNs to accommodate *robustness* objectives to ensure that produced eyeglasses fool face-recognition systems in different imaging conditions (e.g., lighting, angle) and even despite the deployment of specific defenses; *inconspicuousness* objectives, so that the eyeglasses will not arouse the suspicion of human onlookers; and *scalability* objectives requiring that relatively few adversarial objects are sufficient to fool DNNs in many contexts. We show that AGNs can be used to target two DNN-based face-recognition algorithms that achieve human-level accuracy—VGG [159] and OpenFace [3]—and output eyeglasses that enable an attacker to either evade recognition or to impersonate a specific target, while meeting these additional objectives. To demonstrate that AGNs can be effective in contexts other than face recognition, we also train AGNs to fool a classifier designed to recognize handwritten digits and trained on the MNIST dataset [116].

In addition to illustrating the extensibility of AGNs to various types of objectives, these demonstrations highlight two additional features that, we believe, are significant advances. First, AGNs are *flexible* in that an AGN can train a generator to produce adversarial instances with only vaguely specified characteristics. For example, we have no way of capturing inconspicuousness mathematically; rather, we can specify it only using labeled instances. Still, AGNs can be trained to produce new and convincingly

inconspicuous adversarial examples. Second, AGNs are *powerful* in generating adversarial examples that perform better than those produced in previous efforts using more customized techniques. For example, though some of the robustness and inconspicuousness objectives we consider here were also considered in prior work, the adversarial instances produced by AGNs perform better (e.g., ∼70% vs. 31% average success rate in impersonation) and accommodate other objectives (e.g., robustness to illumination changes). AGNs enable attacks that previous methods did not.

We next describe the AGN framework (Sec. 5.2), and its instantiation against face-recognition DNNs (Sec. 5.3). Then, we evaluate the effectiveness of AGNs, including with physically realized attacks and a user study to examine inconspicuousness (Sec. 5.4). Finally, we discuss our work and conclude (Sec. 5.5).

5.2 A Novel Attack Against DNNs

In this section, we describe the AGN framework for attacking DNNs. We define our threat model in Sec. 5.2.1, discuss the challenges posed by vaguely specified objectives in Sec. 5.2.2, provide background on GANs in Sec. 5.2.3, and describe the attack framework in Sec. 5.2.4.

5.2.1 Threat Model

Similarly to Chaps. 3–4, we assume an adversary who gains access to an already trained DNN (e.g., one trained for face recognition), and may only alter the inputs to trick the DNN into misclassifying. We consider targeted attacks (e.g., to *impersonate* another subject enrolled in a face-recognition system) and *untargeted* attacks (e.g., to *dodge* recognition by a face-recognition system). We mainly consider white-box attacks, but we also demonstrate black-box attacks based on transferability from surrogate models, in Sec. 5.4.4.

The framework we propose supports attacks that seek to satisfy a variety of objectives, such as maximizing the DNN's confidence in the target class in impersonation attacks and crafting perturbations that are inconspicuous. Maximizing the confidence in the target class is especially important in scenarios where strict criteria may be used in an attempt to ensure security—for instance, scenarios when the confidence must be above a threshold, as is used to to prevent false positives in face-recognition systems [87]. While inconspicuousness may not be necessary in certain scenarios (e.g., unlocking a mobile device via face recognition), attacks that are not inconspicuous could easily be ruled out in some safety-critical scenarios (e.g., when human operators monitor face-recognition systems at airports [203]).

5.2.2 Vaguely Specified Objectives

In practice, certain objectives, such as inconspicuousness, may elude precise specification. In early stages of our work, while attempting to produce eyeglasses to fool face recognition, we attempted multiple ad-hoc approaches to enhance the inconspicuousness of the eyeglasses in comparison to the attacks shown in Chap. 3, with limited

success. For instance, starting from solid-colored eyeglasses in either of the RGB or HSV color spaces, we experimented with algorithms that would gradually adjust the colors until evasion was achieved, while fixing one or more of the color channels. We also attempted to use Compositional Pattern-Producing Neural Networks [202] combined with an evolutionary algorithm to produce eyeglasses with symmetric or repetitive patterns. These approaches had limited success both at capturing inconspicuousness (e.g., real eyeglasses do not necessarily have symmetric patterns) and at evasion, or failed completely. Thus, instead of pursuing ad hoc approaches to formalize properties that may be insufficient or unnecessary for inconspicuousness, in this work we achieve inconspicuousness via a general framework that models inconspicuous eyeglasses based on many examples thereof, while simultaneously achieving additional objectives, such as evasion.

5.2.3 Generative Adversarial Networks (GANs)

Our attacks build on GANs [65] to create accessories (specifically, eyeglasses) that closely resemble real ones. GANs provide a framework to train a neural network, termed the *generator* ($\mathbb{G}$), to generate data that belongs to a distribution (close to the real one) that underlies a target dataset. $\mathbb{G}$ maps samples from a distribution, Z, that we know how to sample from (such as $[-1,1]^d$, i.e., d-dimensional vectors of reals between -1 and 1) to samples from the target distribution.

To train $\mathbb{G}$, another neural network, called the discriminator ($\mathbb{D}$), is used. $\mathbb{D}$'s objective is to discriminate between real and generated samples. Thus, training can be conceptualized as a game with two players, $\mathbb{D}$ and $\mathbb{G}$, in which $\mathbb{D}$ is trained to emit 1 on real examples and 0 on generated samples, and $\mathbb{G}$ is trained to generate outputs that are (mis)classified as real by $\mathbb{D}$. In practice, training proceeds iteratively and alternates between updating the parameters of $\mathbb{G}$ and $\mathbb{D}$ via back-propagation. $\mathbb{G}$ is trained to minimize the following function:

$$Loss_{\mathbb{G}}(Z,\mathbb{D}) = \sum_{z\in Z} \log\Big(1-\mathbb{D}\big(\mathbb{G}(z)\big)\Big) \tag{5.1}$$

$Loss_G$ is minimized when $\mathbb{G}$ misleads $\mathbb{D}$ (i.e., $D(G(z))$ is 1). $\mathbb{D}$ is trained to maximize the following function:

$$Gain_{\mathbb{D}}(\mathbb{G},Z,data) = \sum_{x\in data} \log\Big(\mathbb{D}\big(x\big)\Big) + \sum_{z\in Z} \log\Big(1-\mathbb{D}\big(\mathbb{G}(z)\big)\Big) \tag{5.2}$$

$Gain_D$ is maximized when $\mathbb{D}$ emits 1 on real samples and 0 on all others.

Several GAN architectures and training methods have been proposed, we build on Deep Convolutional GANs [164].

5.2.4 Attack Framework

Except for a few exceptions [15, 161, 234], in traditional evasion attacks against DNNs the attacker directly alters benign inputs to maximize or minimize a pre-defined function related to the desired misclassification (see Chap. 2). Differently from previous

attacks, we propose to train neural networks to generate outputs that can be used to achieve desired evasions (among other objectives), instead of iteratively tweaking benign inputs to become adversarial.

More specifically, in our AGN framework, we propose to train neural networks to generate images of artifacts (e.g., eyeglasses) that would lead to misclassification. We require that the artifacts generated by these neural networks resemble a reference set of artifacts (e.g., real eyeglass designs), as a means to satisfy an objective that is hard to specify precisely (e.g., inconspicuousness). Similarly to GANs, AGNs are adversarially trained against a discriminator to learn how to generate realistic images. Differently from GANs, AGNs are also trained to generate (adversarial) outputs that can mislead given neural networks (e.g., neural networks designed to recognize faces).

Formally, three neural networks comprise an AGN: a generator, $\mathbb{G}$; a discriminator, $\mathbb{D}$; and a pre-trained DNN whose classification function is denoted by $\mathbb{F}$. When given an input x to the DNN, $\mathbb{G}$ is trained to generate outputs that fool $\mathbb{F}$ and are inconspicuous by minimizing[1]

$$Loss_{\mathbb{G}}(Z, \mathbb{D}) - \kappa \cdot \sum_{z \in Z} Loss_{\mathbb{F}}(x + \mathbb{G}(z)) \tag{5.3}$$

We define $Loss_G$ in the same manner as in Eqn. 5.1; minimizing it aims to generate real-looking (i.e., inconspicuous) outputs that mislead $\mathbb{D}$. $Loss_{\mathbb{F}}$ is a loss function defined over the DNN's classification function that is maximized when training $\mathbb{G}$ (as $-Loss_{\mathbb{F}}$ is minimized). The definition of $Loss_{\mathbb{F}}$ depends on whether the attacker aims to achieve an untargeted misclassification or a targeted one. For untargeted attacks, we use:

$$Loss_{\mathbb{F}}(x + \mathbb{G}(z)) = \sum_{i \neq x} \mathbb{F}_{c_i}(x + \mathbb{G}(z)) - \mathbb{F}_{c_x}(x + \mathbb{G}(z))$$

while for targeted attacks we use:

$$Loss_{\mathbb{F}}(x + \mathbb{G}(z)) = \mathbb{F}_{c_t}(x + \mathbb{G}(z)) - \sum_{i \neq t} \mathbb{F}_{c_i}(x + \mathbb{G}(z))$$

where $\mathbb{F}_c(\cdot)$ is the DNN's output for class c (i.e., the estimated probability of class c in case of a *softmax* activation in the last layer). By maximizing $Loss_{\mathbb{F}}$, for untargeted attacks, the probability of the correct class c_x decreases; for targeted attacks, the probability of the target class c_t increases. We chose this definition of $Loss_{\mathbb{F}}$ because we empirically found that it causes AGNs to converge faster than $Loss_{cw}$ [31], or loss functions defined via cross entropy, as used in Chap. 3. κ is a parameter that balances the two objectives of $\mathbb{G}$; we discuss it further below.

As part of the training process, $\mathbb{D}$'s weights are updated to maximize $Gain_D$, defined in Eqn. 5.2, to tell apart realistic and generated samples. In contrast to $\mathbb{D}$ and $\mathbb{G}$, $\mathbb{F}$'s weights are unaltered during training (as attacks should fool the same DNN at inference time).

The algorithm for training AGNs is provided in Alg. 4. The algorithm takes as input a set of benign examples (X), a pre-initialized generator and discriminator, a

[1] We slightly abuse notation by writing $x+r$ to denote an image x that is modified by a perturbation r. In practice, we use a mask and set the values of x within the masked region to the exact values of r.

Algorithm 4: AGN training

Input : X, $\mathbb{G}$, $\mathbb{D}$, $\mathbb{F}$, *dataset*, Z, N_e, s_b, $\kappa \in \{0, 1\}$
Output: Adversarial $\mathbb{G}$

for $e \leftarrow 1$ **to** N_e **do**
  create *batches* of size s_b from *dataset*;
  for *batch* $\in$ *batches* **do**
    $z \leftarrow s_b$ samples from Z;
    $gen \leftarrow \mathbb{G}(z)$;
    $batch \leftarrow concat(gen, batch)$;
    if *even iteration* **then** `// update` $\mathbb{D}$
      update $\mathbb{D}$ by backpropagating $\frac{\partial Gain_{\mathbb{D}}}{\partial batch}$;
    else `// update` $\mathbb{G}$
      if $\mathbb{F}$ *fooled* **then return** $\mathbb{G}$;
      $d_1 \leftarrow -\frac{\partial Gain_{\mathbb{D}}}{\partial gen}$;
      $x \leftarrow s_b$ sample images from X;
      $x \leftarrow x + gen$;
      Compute forward pass $\mathbb{F}(x)$;
      $d_2 \leftarrow \frac{\partial Loss_{\mathbb{F}}}{\partial gen}$;
      $d_1, d_2 \leftarrow normalize(d_1, d_2)$;
      $d \leftarrow \kappa \cdot d_1 + (1 - \kappa) \cdot d_2$;
      update $\mathbb{G}$ via backpropagating d;

neural network to be fooled, a dataset of real examples (which the generator's output should resemble; in our case this is a dataset of eyeglasses), a function for sampling from $\mathbb{G}$'s latent space (Z), the maximum number of training epochs (N_e), the batch size s_b, and $\kappa \in [0, 1]$. The result of the training process is an *adversarial generator* that creates outputs (e.g., eyeglasses) that fool $\mathbb{F}$. In each training iteration, either $\mathbb{D}$ or $\mathbb{G}$ is updated using a subset of the data chosen at random. $\mathbb{D}$'s weights are updated via gradient ascent to increase $Gain_D$; $\mathbb{G}$'s weights are updated via gradient descent to minimize Eqn. 5.3. To balance the generator's two objectives, the gradients from $Gain_D$ and $Loss_{\mathbb{F}}$ are normalized to the lower Euclidean norm of the two, and then combined into a weighted average controlled by κ. When κ is closer to zero, more weight is given to fooling $\mathbb{F}$ and less to making the output of $\mathbb{G}$ realistic. Conversely, setting κ closer to one places more weight on increasing the resemblance between $\mathbb{G}$'s output and real examples. Training ends when the maximum number of training epochs is reached, or when $\mathbb{F}$ is fooled, i.e., when impersonation or dodging is achieved.

5.3 AGNs that Fool Face Recognition

We next describe how we trained AGNs to generate inconspicuous, adversarial eyeglasses that can mislead state-of-the-art DNNs trained to recognize faces. To do so, we (1) collect a dataset of real eyeglasses; (2) select the architecture of the generator and the discriminator, and instantiate their weights; (3) train DNNs that can evaluate the attacks; and (4) set the parameters for the attacks.

Figure 5.1: Examples of raw images of eyeglasses that we collected (left) and their synthesis results (right).

5.3.1 Collecting a Dataset of Eyeglasses

A dataset of real eyeglass-frame designs is necessary to train the generator to create real-looking attacks. We collected such a dataset using Google's search API.[2] To collect a variety of designs, we searched for "eyeglasses" and synonyms (e.g., "glasses," "eyewear"), sometimes modified by an adjective, including colors (e.g., "brown," "blue"), trends (e.g., "geek," "tortoise shell"), and brands (e.g., "Ralph Lauren," "Prada"). In total, we made 430 unique API queries and collected 26,520 images.

The images we collected were not of only eyeglasses; e.g., we found images of cups, vases, and logos of eyeglass brands. Some images were of eyeglasses worn by models or on complex backgrounds. Such images would hinder the training process. Hence, we trained a classifier to detect and keep only images of eyeglasses over white backgrounds and not worn by models. Using 250 hand-labeled images, we trained a classifier that identified such images with 100% precision and 65% recall. After applying it to all the images in the dataset, 8,340 images remained. Manually examining a subset of these images revealed no false positives.

Using images from this dataset, we could train a generator that can emit eyeglasses of different patterns, shapes, and orientations. However, variations in shape and orientation made such eyeglasses difficult to efficiently align to face images while running Alg. 4. Therefore, we preprocessed the images in the dataset and transferred the patterns from their frames to a fixed shape (the same used in Chap. 3, see Fig. 3.1), which we could then easily align to face images. We then trained the generator to emit images of eyeglasses with this particular shape, but with different colors and textures. To transfer the colors and textures of eyeglasses to a fixed shape, we thresholded the images to detect the areas of the frames. (Recall that the backgrounds of the images were white.) We then used Efros and Leung's texture-synthesis technique to synthesize the texture from the frames onto the fixed shape [54]. Fig. 5.1 shows examples. Since the texture synthesis process is nondeterministic, we repeated it twice per image. At the end of this process, we had 16,680 images for training.

Since physical realizability is a requirement for our attacks, it was important that the generator emitted images of eyeglasses that are *printable*. In particular, the colors of the eyeglasses needed to be within the range our commodity printer (Epson XP-830) could print. Therefore, we mapped the colors of the eyeglass frames in the dataset into the color gamut of our printer. To model the color gamut, we printed an image containing all 2^{24} combinations of RGB triplets, captured a picture of that image, and computed the convex hull of all the RGB triplets in the captured image. To make an image of eyeglasses printable, we mapped each RGB triplet in the image to the closest RGB triplet found within the convex hull.

[2] `https://developers.google.com/custom-search/`

5.3.2 Pretraining the Generator and the Discriminator

When training GANs, it is desirable for the generator to emit sharp, realistic, diverse images. Emitting only a small set of images would indicate the generator's function does not approximate the underlying distribution well. To achieve these goals, and to enable efficient training, we chose the Deep Convolutional GAN, a minimalistic architecture with a small number of parameters [164]. In particular, this architecture is known for its ability to train generators that can emit sharp, realistic images.

We then explored a variety of options for the generator's latent space and output dimensionality, as well as the number of weights in both $\mathbb{G}$ and $\mathbb{D}$ (via adjusting the depth of filters). We eventually found that a latent space of $[-1, 1]^{25}$ (i.e., 25-dimensional vectors of real numbers between -1 and 1), and output images of 64×176 pixels produced the best-looking, diverse results. The final architectures of $\mathbb{G}$ and $\mathbb{D}$ are reported in Fig. 5.2.

To ensure that attacks converged quickly, we initialized $\mathbb{G}$ and $\mathbb{D}$ to a state in which the generator can already produce real-looking images of eyeglasses. To do so, we pretrained $\mathbb{G}$ and $\mathbb{D}$ for 200 epochs and stored them to initialize later runs of Alg. 4.[3] Moreover, we used Salimans et al.'s recommendation and trained $\mathbb{D}$ on *soft labels* [177]. Specifically, we trained $\mathbb{D}$ to emit 0 on samples originating from the generator, and 0.9 (instead of 1) on real examples. Fig. 5.3 presents a couple of eyeglasses emitted by the pretrained generator.

5.3.3 DNNs for Face Recognition

We evaluated our attacks against four DNNs of two architectures. Two of the DNNs were built on the *Visual Geometry Group (VGG)* neural network [159]—the same one used in Chap. 3. The other two DNNs were built on the OpenFace neural network, which uses the Google FaceNet architecture [3]. OpenFace's main design consideration is to provide high accuracy with low training and prediction times so that the DNN can be deployed on mobile and IoT devices. Hence, the DNN is relatively compact, with 3.74 million parameters, but nevertheless achieves near-human accuracy on the LFW benchmark (92.92%).

We trained one small and one large face-recognition DNN for each architecture, using the same data used in Chap. 3 (see Sec. 3.2.1). We call the small DNNs of the VGG and OpenFace architectures VGG10 and OF10, as they were trained to recognize 10 subjects. The large DNNs, termed VGG143 and OF143, were trained to recognize 143 subjects. In all cases, three of the subjects that the DNNs were trained to recognize (the author of the thesis and two collaborators) were available to us locally for evaluating the attacks in the physical environment. We refer to them by S_A, S_B, and S_C, as in Sec. 3.3.2.

Training the VGG Networks As a reminder, the original VGG network takes a 224×224 aligned face image as input and produces a highly discriminative face descriptor (i.e., vector representation of the face) of 4096 dimensions. Two descriptors of images of the same person are designed to be closer to each other in Euclidean

[3] For training, we used the Adam optimizer [99] and set the learning rate to 2×10^{-4}, the batch size to 260, β_1 to 0.5, and β_2 to 0.999.

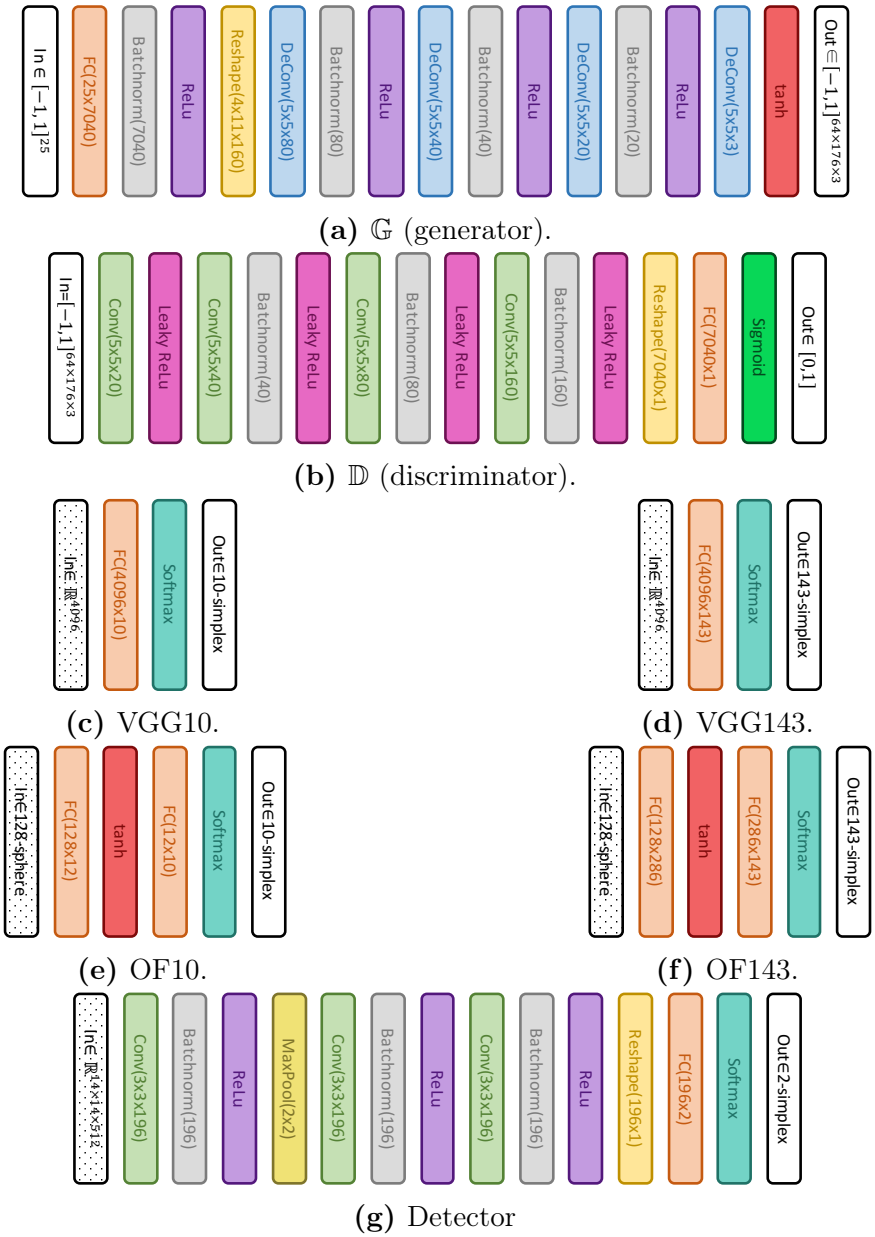


Figure 5.2: Architectures of the neural networks used in this work. Inputs that are intermediate (i.e., received from feature-extraction DNNs) have dotted backgrounds. *Deconv* refers to transposed convolution, and *FC* to fully connected layer. N-simplex refers to the set of probability vectors of N dimensions, and the 128-sphere denotes the set of real 128-dimensional vectors lying on the Euclidean unit sphere. All convolutions and deconvolutions in $\mathbb{G}$ and $\mathbb{D}$ have strides and paddings of two. The detector's convolutions have strides of two and padding of one. The detector's max-pooling layer has a stride of two.

space than two descriptors of different people's images. We again used transfer learning [229] to train DNNs for face recognition. Specifically, we used the descriptors to train two simple neural networks that map face descriptors to probabilities over the set of identities (thus the original DNNs effectively act as feature extractors).

Figure 5.3: Examples of eyeglasses emitted by the generator (left) and similar eyeglasses from the training set (right).

The architectures of the VGG-derived neural networks are provided in Fig. 5.2. They consist of fully connected layers (i.e., linear separators) connected to a *softmax* layer that turns the linear separators' outputs into probabilities. We trained the networks by minimizing cross-entropy loss ($Loss_{ce}$) [64]. After training, we connected the trained neural networks to the original VGG network to construct end-to-end DNNs that map face images to identities.

An initial evaluation of VGG10 and VGG143 showed high performance. To verify that the DNNs cannot be easily misled, we tested them against naïve attacks by attaching eyeglasses emitted by the pretrained (non-adversarial) generator to test images. We found that impersonations of randomly picked targets are unlikely—they occur with 0.79% chance for VGG10 and <0.01% for VGG143. However, we found that dodging would succeed with non-negligible chance: 7.81% of the time against VGG10 and 26.87% against VGG143. We speculated that this was because the training samples for some subjects never included eyeglasses. To make the DNNs more robust, we augmented their training data following adversarial training techniques [113]: for each image initially used in training we added two variants with generated eyeglasses attached. We also experimented with using more variants but found no additional improvement. Also following Kurakin et al., we included 50% raw training images and 50% augmented images in each batch during training [113].

Evaluating VGG10 and VGG143 on held-out test sets after training, we found that they achieved 100% and 98% accuracy, respectively. In addition, the success of naïve dodging was at most 4.60% and that of impersonation was below 0.01%. Finally, to maintain a high level of security, it is important to minimize the DNNs' false positives [87]. Once again, we do so by setting a criteria on the DNNs' output to decide when it should be accepted. We were able to find thresholds for the probabilities emitted by VGG10 and VGG143 such that their accuracies remained 100% and 98%, while the FPRs of both DNNs were 0%. The performance of the DNNs is reported in Table 5.1.

Training the OpenFace Networks The original OpenFace network takes a 96×96 aligned face image as input and outputs a face descriptor of 128 dimensions. Similar to the VGG networks, the descriptors of images of the same person are close in Euclidean space, while the descriptors of different people's images are far. Unlike VGG, the OpenFace descriptors lie on a unit sphere.

We again used transfer learning to train the OpenFace networks. We first attempted to train neural networks that map the OpenFace descriptors to identities using architectures similar to the ones used for the VGG DNNs. We found these neural networks to achieve competitive accuracies. Similarly to the VGG DNNs, they were also vulnerable to naïve dodging attempts, but unlike the VGG DNNs, straightforward data augmentation did not improve their robustness. We believe this may stem from limitations of classifying data on a sphere using linear separators.

Model	*acc.*	*SR naïve dodge*	*SR naïve impers.*	*thresh.*	*TPR*	*FPR*
VGG10	100%	3%	0%	0.92	100%	0%
VGG143	98%	5%	0%	0.82	98%	0%
OF10	100%	14%	1%	0.55	100%	0%
OF143	86%	22%	<1%	0.91	59%	2%

Table 5.1: Performance of the face-recognition DNNs. We report the accuracy, the success rate (SR) of naïve dodging and impersonation (likelihood of naïve attackers to be misclassified arbitrarily or as *a priori* chosen targets), the threshold to balance correct and false classifications, the true-positive rate (TPR; how often the correct class is assigned a probability above the threshold), and the false-positive rate (FPR; how often a wrong class is assigned a probability above the threshold).

To improve the robustness of the DNNs, we increased their depth by prepending a fully connected layer followed by a hyperbolic-tangent (*tanh*) layer (see Fig. 5.2). This architecture was chosen as it performed the best out of different ones we experimented with. We also increased the number of images we augmented in training to 10 (per image in the training set) for OF10 and to 100 for OF143. The number of images augmented was selected such that increasing it did not further improve robustness against naïve attacks. Similarly to the VGG networks, we trained with about 40 images per subject, and included 50% raw images and 50% augmented images in training batches.

We report the performance of the networks in Table 5.1. OF10 achieved 100% accuracy, while OF143 achieved 85.50% accuracy (comparable to Amos et al.'s finding [3]). The OpenFace DNNs were more vulnerable to naïve attacks than the VGG DNNs. For instance, OF10 failed against 14.10% of the naïve dodging attempts and 1.36% of the naïve impersonation attempts. We believe that the lower accuracy and higher susceptibility of the OpenFace DNNs compared to the VGG DNNs may stem from the limited capacity of the OpenFace network induced by the small number of parameters.

Training an Attack Detector In addition to the face-recognition DNNs, we trained a DNN to detect attacks that target the VGG networks following the proposal of Metzen et al. [139] (see Sec. 2.3). We chose this detector because at the time of the work it was one of the most effective detectors against imperceptible adversarial examples [30, 139]. We focused on VGG DNNs because no detector architecture was proposed for detecting attacks against OpenFace-like architectures. To mount a successful attack when a detector is deployed, it is necessary to simultaneously fool the detector and the face-recognition DNN.

We used the architecture proposed by Metzen et al. (see Fig. 5.2). For best performance, we attached the detector after the fourth max-pooling layer of the VGG network. To train the detector, we used 170 subjects from the original dataset used by Parkhi et al. for training the VGG network [159]. For each subject we used 20 images for training. For each training image, we created a corresponding adversarial image that evades recognition. We trained the detector for 20 epochs using the Adam opti-

mizer with the training parameters set to standard values (learning rate = 1×10^{-4}, $\beta_1 = 0.99$, $\beta_2 = 0.999$) [99]. At the end of training, we evaluated the detector on 20 subjects who were not used in training, finding that it had 100% recall and 100% precision.

5.3.4 Implementation Details

We used the Adam optimizer to update the weights of $\mathbb{D}$ and $\mathbb{G}$ when running Alg. 4. As in pretraining, β_1 and β_2 were set to 0.5 and 0.999. We ran grid search to set κ and the learning rate, and found that a $\kappa = 0.25$ and a learning rate of 5×10^{-5} gave the best tradeoff between success in fooling the DNNs, inconspicuousness, and the algorithm's run time. The number of epochs was limited to at most one, as we found that the results the algorithm returned when running longer were not inconspicuous.

The majority of our work was implemented in MatConvNet, a MATLAB toolbox for convolutional neural networks [211]. The OpenFace DNN was translated from the original implementation in Torch to MatConvNet. We released the implementation online: `https://github.com/mahmoods01/agns`.

5.4 Evaluation

We extensively evaluated AGNs as an attack method. In Sec. 5.4.1 we show that AGNs reliably generate successful dodging and impersonation attacks in a digital environment, even when a detector is used to prevent them. We show in Sec. 5.4.2 that these attacks can also be successful in the physical domain. In Sec. 5.4.3, we demonstrate universal dodging, i.e., generating a small number of eyeglasses that many subjects can use to evade recognition. We test in Sec. 5.4.4 how well our attacks transfer between models. Table 5.2 summarizes the combinations of domains the attacks were performed in, the attack types considered, and what objectives were aimed or tested for in each experiment. While more combinations exist, we attempted to experiment with the most interesting combinations under computational and manpower constraints. (For example, attacks in the physical domain require significant manual effort, and universal attacks require testing with more subjects than is feasible to test with in the physical domain.) In Sec. 5.4.5 we demonstrate that AGNs can generate eyeglasses that are inconspicuous to human participants in a user study. Finally, in Sec. 5.4.6 we show that AGNs are applicable to areas other than face recognition (specifically, by fooling a digit-recognition DNN).

5.4.1 Attacks in the Digital Domain

In contrast to physically realized attacks, an attacker in the digital domain can exactly control the input she provides to DNNs, since the inputs are not subject to noise added by physically realizing the attack or capturing the image with a camera. Therefore, our first step again is to verify that the attacker can successfully fool the DNNs in the digital domain, as failure in the digital domain implies failure in the physical domain.

	Domain		*Type*		*Objectives*							
						Robustness vs.						
Sec.	*Dig.*	*Phys.*	*Untarg.*	*Targ.*	*Incons.*	*Aug.*	*Detect.*	*Print.*	*Pose*	*Lum.*	*Univ.*	*Trans.*
5.4.1	✓		✓		✓	✓		✓				
	✓		✓		✓	✓	✓	✓				
	✓			✓	✓	✓		✓				
	✓			✓	✓	✓	✓	✓				
5.4.2		✓	✓		✓	✓		✓	✓	✓		
		✓		✓	✓	✓		✓	✓	✓		
5.4.3	✓		✓		✓	✓		✓			✓	
5.4.4	✓		✓		✓	✓		✓				✓
	✓		✓		✓	✓		✓			✓	✓
		✓	✓		✓	✓		✓	✓	✓		✓

Table 5.2: For each experimental section, we mark the combinations of domains (digital or physical) in which the attack was tested, the attack types (untargeted or targeted) tested, and the objectives of the attack, chosen from inconspicuousness, robustness (against training-data augmentation, detection, printing noise, pose changes, and luminance changes), universality, and transferability. Note that while we did not explicitly design the attacks to transfer between architectures, we found that they transfer relatively well; see Sec. 5.4.4.

Experiment Setup To evaluate the attacks in the digital domain, we followed a similar approach to Sec. 3.3.1. Particularly, we selected a set of subjects for each DNN from the subjects the DNNs were trained on: 20 subjects selected at random for VGG143 and OF143 and all ten subjects for VGG10 and OF10. In impersonation attacks, the targets were chosen at random. To compute the uncertainty in our estimation of success, we repeated each attack three times, each time using a different image of the attacker.

As baselines for comparison, we evaluated three additional attacks using the same setup. The first attack is the one presented in Chap. 3. We refer to it by *CCS16*, in reference to the conference the attack was published in. Similarly to before, We ran the attack up to 300 iterations, starting from solid colors, and clipping the colors of the eyeglasses to the range $[0, 1]$ after each iteration to ensure that they lie in a valid range. The second attack is the *PGD* attack of Madry et al. [129], additionally constrained to perturb only the area covered by the eyeglasses. Specifically, we started from eyeglasses with solid colors and iteratively perturbed them for up to 100 iterations, while clipping the perturbations to have max-norm (i.e., L_∞-norm) of at most 0.12. We picked 100 and 0.12 as the maximum number of iterations and max-norm threshold, respectively, as these parameters led to the most powerful attack in prior work [129]. The *PGD* attack can be roughly seen as a special case of the *CCS16* attack, as the two attacks follow approximately the same approach except that *PGD* focuses on a narrower search space by clipping perturbations more aggressively to decrease their

perceptibility. As we show below, the success rate of *PGD* was significantly lower than of *CCS16*, suggesting that the clipping *PGD* performs may be too aggressive for this application. Therefore, we evaluated a third attack, denoted $\widehat{CCS16}$, in which we clipped perturbations, but did so less aggressively than *PGD*. In $\widehat{CCS16}$, we set the number of iterations to 300, as in the *CCS16* attack, and clipped perturbations to have max-norm of at most 0.47. We selected 0.47 as the max-norm threshold as it is the lowest threshold that led to success rates at fooling face recognition that are comparable to *CCS16*. In other words, this threshold gives the best chance of achieving inconspicuousness without substantially sacrificing the success rates. To maximize the success rates of the three attacks, we optimized only for the evasion objectives (defined via $Loss_{ce}$), and ignored other objectives that are necessary to physically realize the attacks (specifically, generating colors that can be printed by minimizing *NPS*, and ensuring smooth transitions between neighboring pixels by minimizing *TV*, see Sec. 3.2).

To test whether a detector would prevent attacks based on AGNs, we selected all ten subjects for VGG10 and 20 random subjects for VGG143, each with three images per subject. We then tested whether dodging and impersonation can be achieved while simultaneously evading the detector. To fool the detector along with the face-recognition DNNs, we slightly modified the objective from Eqn. 5.3 to optimize the adversarial generator such that the detector's loss is increased. In particular, the loss function we used was the difference between the probabilities of the correct class (either "adversarial" or "non-adversarial" input) and the incorrect class. As the vanilla *PGD*, *CCS16*, and $\widehat{CCS16}$ attacks either failed to achieve success rates comparable to AGNs or produced more conspicuous eyeglasses (Sec. 5.4.5), we did not extend them to fool the detector.

We again evaluated attacks via success rates. For dodging, we measured the percentage of attacks in which the generator emitted eyeglasses that (1) led the image to be misclassified (i.e., the most probable class was not the attacker), and (2) kept the probability of the correct class below 0.01 (much lower than the thresholds set for accepting any of the DNNs' classifications; see Table 5.1). For impersonation, we considered an attack successful if the attacker's image was classified as the target with probability exceeding 0.92, the highest threshold used by any of the DNNs. When using the detector, we also required that the detector deemed the input non-adversarial with probability higher than 0.5.

Experiment Results Table 5.3 summarizes the results of the digital-environment experiments (without using an attack detector). All dodging attempts using AGNs succeeded; Fig. 5.4 shows an example. As with dodging, all impersonation attempts using AGNs against the small DNNs (VGG10 and OF10) succeeded. A few attempts against the larger DNNs failed, suggesting that inconspicuous impersonation attacks may be more challenging when the DNN recognizes many subjects, although attacks succeeded at least 88% of the time.

Differently from AGNs, the success of *PGD* was limited—its success rates at dodging ranged from 37% to 85%, and its success rates at impersonation were below 13%. This suggests that the clipping performed by *PGD* may be too aggressive. The *CCS16* and $\widehat{CCS16}$ attacks achieved success rates comparable to AGNs at both dodging and

	Dodging				*Impersonation*			
Model	*AGNs*	*PGD*	$\widehat{CCS16}$	*CCS16*	*AGNs*	*PGD*	$\widehat{CCS16}$	*CCS16*
VGG10	100±0%	37±10%	100±0%	100±0%	100±0%	10±5%	100±0%	100±0%
VGG143	100±0%	53±9%	100±0%	100±0%	88±5%	3±2%	82±5%	98±2%
OF10	100±0%	43±11%	100±0%	100±0%	100±0%	13±7%	87±9%	100±0%
OF143	100±0%	85±7%	100±0%	100±0%	90±4%	11±4%	78±6%	88±4%

Table 5.3: Results of attacks in the digital environment. We report the the mean success rate of attacks and the standard error when fooling the facial-recognition DNNs.

Figure 5.4: An example of digital dodging. Left: An image of actor Owen Wilson (from the PubFig dataset [111]), correctly classified by VGG143 with probability 1.00. Right: Dodging against VGG143 using AGN's output (probability assigned to the correct class <0.01).

impersonation. Unlike for AGNs, in this set of experiments the high success rates of the *CCS16* and $\widehat{CCS16}$ attacks were achieved by only attempting to dodge or impersonate (i.e., fool the classifier), while ignoring other objectives, such as generating colors which can be printed (via minimizing *NPS*). For physically realized attacks, evaluated in Sec. 5.4.2, satisfying these additional objectives is necessary for the *CCS16* and $\widehat{CCS16}$ attacks to succeed; there, however, measurements suggest that AGNs capture the inconspicuousness and realizability objectives more effectively.

As shown in Table 5.4 using a detector did not thwart the AGN attacks: success rates for dodging and impersonation were similar to when a detector was not used. However, using a detector reduced the inconspicuousness of attacks (see Sec. 5.4.5).

We further tested whether attackers can be more successful by using eyeglasses of different shapes: we trained AGNs to generate eyeglasses of six new shapes and tested them against VGG143 and OF143. Three of the new shapes achieved comparable performance to the original shape (shown in Fig. 5.4), but the overall success rates did not improve. Nevertheless, it would be useful to explore whether using variety of eyeglass shapes can enhance the inconspicuousness of attacks in practice.

5.4.2 Attacks in the Physical Domain

Attackers in the physical domain do not have complete control over the DNN's input: Slight changes in the attacker's pose, expression, distance from the camera, and illumination may dramatically change the concrete values of pixels. Practical attacks need

Model	*Dodging*	*Impersonation*
VGG10	100±0%	100±0%
VGG143	100±0%	90±4%

Table 5.4: The mean success rates and standard errors of dodging and impersonation using AGNs when simultaneously fooling facial-recognition DNNs and a detector.

to be robust against such changes. We took three additional measures to make the attacks more robust.

First, to train adversarial generators that emit images of eyeglasses that lead to more than one of the attacker's images to be misclassified, we used multiple images of the attacker in training the generator. Namely, we set X in Alg. 4 to be a collection of the attacker's images. As a result, the generators learned to maximize $Loss_{\mathbb{F}}$ for different images of the attacker.

Second, to make the attacks robust to changes in pose, we trained the adversarial generator to minimize $Loss_{\mathbb{F}}$ over multiple images of the attacker wearing the eyeglasses. To align the eyeglasses to the attacker's face, we created and printed a 3d model of eyeglasses with frames that have the same silhouette as the 2d eyeglasses emitted by the generator (using code from GitHub [28]). We added tracking markers—specifically positioned green dots—to the 3d-printed eyeglasses. The attacker wore the eyeglasses when capturing training data for the generator. We then used the markers to find a projective alignment, θ_x, of the eyeglasses emitted by the generator to the attacker's pose in each image. The generator was subsequently trained to minimize $Loss_{\mathbb{F}}\big(x + \theta_x(G(z))\big)$ for different images of the attacker ($x \in X$).

Third, to achieve robustness to varying illumination conditions, we modeled how light intensity (luminance) affects eyeglasses and incorporated the models in AGN training. Specifically, we used the Polynomial Texture Maps approach [132] to estimate degree-3 polynomials that map eyeglasses' RGB values under baseline luminance to values under a specific luminance. In the forward pass of Alg. 4, before digitally attaching eyeglasses to an attacker's image of certain luminance, we mapped the eyeglasses' colors to match the image's luminance. In the backward pass, the errors were back-propagated through the polynomials before being back-propagated through the generator to adjust its weights. In this way, the texture-map polynomials enabled us to digitally estimate the effect of lighting on the eyeglasses.

Experiment Setup To evaluate the physically realized attacks, the subjects S_A, S_B, and S_C acted as attackers. Each subject attempted both dodging and impersonation against each of the four DNNs (which were trained to recognize them, among others). The data used for training and evaluating the physically realized attacks were collected from a room with a ceiling light but with no windows on exterior walls using a Canon T4i camera (similarly to Sec. 3.3.2).

In a first set of experiments, we evaluated the attacks under varied poses. To this end, we followed a similar methodology to Sec. 3.3.2. Specifically, to train the adversarial generators, we collected 45 images of each attacker (the set X in Alg. 4) while he or she stood a fixed distance from the camera, kept a neutral expression,

and moved his or her head up-down, left-right, and in a circle. Each generator was trained for at most one epoch, and training stopped earlier if the generator could emit eyeglasses that, for dodging, led the mean probability of the correct class to fall below 0.005, or, for impersonation, led the mean probability of the target class to exceed 0.990. For impersonation, we picked the target at random per attack.

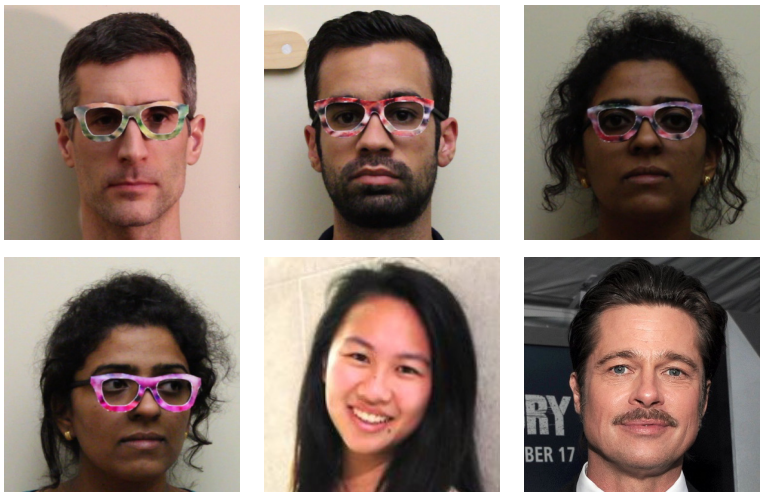
To physically realize the attacks, we printed selected eyeglass patterns created by the generator (again, on Epson Ultra Premium Glossy paper using a commodity Epson XP-830 printer) and affixed them to the 3d-printed eyeglasses. Since each generator can emit a diverse set of eyeglasses, we (digitally) sampled 48 outputs (qualitatively, this amount seemed to capture the majority of patterns that the generators could emit) and kept the most successful one for dodging or impersonation in the digital environment (i.e., the one that led to the lowest mean probability assigned the attacker or the highest mean probability assigned to the target, respectively).

We evaluated the attacks by collecting videos of the attackers wearing the 3d-printed eyeglasses with the adversarial patterns affixed to their front. Again, the attackers were asked to stand a fixed distance from the camera, keep a neutral expression, and move their heads up-down, left-right, and in a circle. We extracted every third frame from each video. This resulted in 75 frames, on average, per attack. We then classified the extracted images using the DNNs targeted by the attacks. For dodging, we measured success by the fraction of frames that were classified as anybody but the attacker, and for impersonation by the fraction of frames that were classified as the target. In some cases, impersonation failed—mainly due to the generated eyeglasses not being realizable, as many of the pixels had extreme values (close to RGB=[0,0,0] or RGB=[1,1,1]). In such cases, we attempted to impersonate another (randomly picked) target.

We measured the head poses (i.e., pitch, yaw, and roll angles) of the attackers in training images using a state-of-the-art tool [14]. On average, head poses covered 13.01° of pitch (up-down direction), 17.11° of yaw (left-right direction), and 4.42° of roll (diagonal direction). This is similar to the mean difference in head pose between pairs of images randomly picked from the PubFig dataset [111] (11.64° of pitch, 15.01° of yaw, and 6.51° of roll).

As a baseline to compare to, we repeated the dodging and impersonation attempts using our attack from Chap. 3, referred to by *CCS16*. Unlike experiments in the digital domain, we did not evaluate variants of the *CCS16* attack where additional clipping is performed, as our experience in the digital domain showed that clipping harms the success rate of attacks and fails to improve their inconspicuousness (see Sec. 5.4.5).

In a second set of experiments, we evaluated the effects of changes to luminance. To this end, we placed a lamp (with a $150\,W$ incandescent light bulb) about 45° to the left of the attacker, and used a dimmer to vary the overall illuminance between $\sim 110 lx$ and $\sim 850 lx$ (comparable to difference between a dim corridor and a bright chain store interior [12]). We crafted the attacks by training the generator on 20 images of the attacker collected over five equally spaced luminance levels. In training the generator, we used the polynomial texture models as discussed above. For impersonation, we used the same targets as in the first set of experiments. We implemented the eyeglasses following the same procedure as before, then collected 40 video frames per attack, split evenly among the five luminance levels. In these experiments, the attackers again stood a fixed distance from the camera, but did not vary their pose. For this set of



(a) (b) (c)

Figure 5.5: Examples of physically realized attacks. (a) S_A (top) and S_B (bottom) dodging against OF143. (b) S_C impersonating S_E against VGG10. (c) S_B impersonating actor Brad Pitt (by Marvin Lynchard / CC BY 2.0 / cropped from `https://goo.gl/Qnhe2X`) against VGG10.

experiments, we do not compare with our previous algorithm, as it was not designed to achieve robustness to changing luminance, and informal experiments showed that it performed poorly when varying the luminance levels.

Experiment Results Table 5.5 summarizes our results and Fig. 5.5 shows examples of attacks in the physical environment.

In the first set of experiments we varied the attackers' pose. Most dodging attempts succeeded with all video frames misclassified. Even in the worst attempt, 81% of video frames were misclassified. Overall, the mean probability assigned to the correct class was at most 0.40, much below the thresholds discussed in Sec. 5.3.3. For impersonation, one to four subjects had to be targeted before impersonations succeeded, with an average of 68% of video frames (mis)classified as the targets in successful impersonations. In two thirds of these attempts, >20% of frames were misclassified with high confidence (again, using the thresholds from Sec. 5.3.3). This suggests that even a conservatively tuned system would likely be fooled by some attacks.

We found that physical-domain evasion attempts using AGNs were significantly more successful than attempts using the *CCS16* algorithm. The mean success rate of dodging attempts was 45% higher when using AGNs compared to prior work (97% vs. 67%; a paired t-test shows that the difference is statistically significant with $p = 0.03$). The difference in success rates for impersonation was even larger. The mean success rate of impersonation attempts was 126% higher using AGNs compared to prior work (70% vs. 31%; paired t-test shows that the difference is statistically significant with $p < 0.01$). Given these results, we believe that AGNs provide a better approach to test the robustness of DNNs against physical-domain attacks than the *CCS16* algorithm.

The second set of experiments shows that AGNs can generate attacks that are robust to changes in luminance. On average, the dodging success rate was 96% (with

Dodging results					
DNN	*Subject*	*AGNs*	*p*(sub.)	*CCS16*	*AGNs-L*
VGG10	S_A	97%	0.09	98%	100%
	S_B	96%	0.10	100%	100%
	S_C	100%	0.06	0%	93%
VGG143	S_A	100%	<0.01	87%	100%
	S_B	100%	0.03	82%	100%
	S_C	98%	0.17	0%	100%
OF10	S_A	100%	0.01	100%	100%
	S_B	100%	0.01	100%	100%
	S_C	81%	0.40	0%	83%
OF143	S_A	97%	0.05	97%	100%
	S_B	100%	<0.01	99%	100%
	S_C	100%	0.09	36%	75%

Impersonation results								
DNN	*Subject*	*Target*	*Attempts*	*AGNs*	*HC*	*p*(tar.)	*CCS16*	*AGNs-L*
VGG10	S_A	Milla Jovovich	2	88%	0%	0.70	63%	100%
	S_B	Brad Pitt	1	100%	96%	0.98	100%	100%
	S_C	S_E	1	100%	74%	0.93	0%	80%
VGG143	S_A	Ashton Kutcher	2	28%	0%	0.22	0%	0%
	S_B	Daniel Radcliffe	2	3%	0%	0.04	0%	16%
	S_C	Alicia Keys	2	89%	41%	0.73	0%	70%
OF10	S_A	Brad Pitt	2	65%	55%	0.58	43%	100%
	S_B	S_E	1	98%	95%	0.83	67%	38%
	S_C	Brad Pitt	2	28%	23%	0.25	0%	50%
OF143	S_A	Aaron Eckhart	4	99%	25%	0.83	92%	100%
	S_B	Eva Mendes	2	53%	39%	0.67	3%	9%
	S_C	Carson Daly	1	60%	0%	0.41	0%	73%

Table 5.5: Summary of physical realizability experiments. For dodging (top), we report the success rate of AGNs (percentage of misclassified video frames), the mean probability assigned to the correct class (lower is better), the success rate of the *CCS16* attack (Chap. 3), and the success rate of AGNs under luminance levels higher than the baseline luminance level (AGNs-L). For impersonation (bottom), we report the target (S_E is a member of our group, an Asian female in the early 20s), the number of targets attempted until succeeding, the success rate of AGNs (percentage of video frames classified as the target), the fraction of frames classified as the target with high confidence (HC; above the threshold which strikes a good balance between the true and the false positive rate), the mean probability assigned to the target (higher is better), the success rate of the *CCS16* attack, and the success rate under varied luminance levels excluding the baseline level (AGNs-L). Non-adversarial images of the attackers were assigned to the correct class.

most attempts achieving 100% success rate), and the impersonation success rate was 61%. Both are comparable to success rates under changing pose and fixed luminance. To evaluate the importance of modeling luminance to achieve robustness, we measured

the success rate of S_C dodging against the four DNNs *without* modeling luminance effects. This caused the average success rate of attacks to drop from 88% (the average success rate of S_C at dodging when modeling luminance) to 40% (marginally significant according to a t-test, with $p = 0.06$). This suggests that modeling the effect of luminance when training AGNs is essential to achieve robustness to luminance changes.

Last, we built a mixed-effects logistic regression model [160] to analyze how different factors, and especially head pose and luminance, affect the success of physical-domain attacks. In the model, the dependent variable was whether an image was misclassified, and the independent variables accounted for the absolute value of pitch (up-down), yaw (left-right), and roll (tilt) angles of the head in the image (measured with Baltrušaitis et al.'s tool [14]); the luminance level (normalized to a [0,4] range); how close are the colors of the eyeglasses printed for the attack to colors that can be produced by our printer (measured via *NPS*, and normalized to a [0,1] range); the architecture of the DNN attacked (VGG or OpenFace); and the size of the DNN (10 or 143 subjects). The model also accounted for the interaction between angles and architecture, as well as the luminance and architecture.

To train the model, we used all the images we collected to test the attack in the physical domain. The model's R^2 is 0.70 (i.e., it explains 70% of the variance in the data), indicating a good fit. The parameter estimates are shown in Table 5.6. Luminance is not a statistically significant factor—i.e., the DNNs were equally likely to misclassify the images under the different luminance levels we considered. In contrast, the face's pose has a significant effect on misclassification. For the VGG networks, each degree of pitch or yaw away from 0° reduced the likelihood of success by 0.94, on average. Thus, an attacker who faced the camera at a pitch or yaw of ±10° was about 0.53 times less likely to succeed than when directly facing the camera. Differently from the VGG networks, for the OpenFace networks each degree of pitch away from 0° increased the likelihood of success by 1.12, on average. Thus, an attacker facing the camera at a pitch of ±10° was about 3.10 times more likely to succeed than when directly facing the camera. Overall, these results highlight the attacks' robustness to changes in luminance, as well as to small changes in pose away from frontal.

5.4.3 Universal Dodging Attacks

We next show that a small number of adversarial eyeglasses can allow successful dodging for the majority of subjects, even when images of those subjects are not used in training the adversarial generator.

We created the universal attacks by training the generator in Alg. 4 on a set of images of different people. Consequently, the generator learned to emit eyeglasses that caused multiple people's images to be misclassified, not only one person's. We found that when the number of subjects was large, the generator started emitting conspicuous patterns that did not resemble real eyeglasses. For such cases, we used Alg. 5, which builds on Alg. 4 to train several adversarial generators, one per cluster of similar subjects. Alg. 5 uses *k-means++* [8] to create clusters of size s_c. Clustering was performed in Euclidian space using the features extracted from the base DNNs (4096-dimensional features for VGG, and 128-dimensional features for OpenFace; see Sec. 5.3.3). The result was a set of generators that create eyeglasses that, cumulatively, (1) led to the

Factor	log(*odds*)	*odds*	*p-value*
(intercept)	<0.01	1.00	0.96
is.143.subjects.dnn	-0.48	0.62	<0.01
is.openface.dnn	6.34	568.78	<0.01
abs(pitch)	-0.06	0.94	<0.01
abs(yaw)	-0.06	0.94	<0.01
abs(roll)	0.01	1.01	0.80
luminance	0.04	1.04	0.12
non-printability	-1.09	0.34	<0.01
is.openface.dnn:luminance	0.38	1.48	0.14
is.openface.dnn:abs(pitch)	0.18	1.19	<0.01
is.openface.dnn:abs(yaw)	-0.08	0.92	0.06
is.openface.dnn:abs(roll)	-0.62	0.54	<0.01

Table 5.6: Parameter estimates for the logistic regression model. Statistically significant factors are in boldface.

Algorithm 5: Universal attacks (given many subjects)

Input : X, $\mathbb{G}$, $\mathbb{D}$, $\mathbb{F}$, *dataset*, Z, N_e, s_b, κ, s_c
Output: *Gens* `// a set of generators`

Gens $\leftarrow \{\}$;
clusters $\leftarrow$ clusters of size s_c via *k-means++*;
for *cluster* $\in$ *clusters* **do**
  $\hat{\mathbb{G}} \leftarrow$ *Alg1*(*cluster*, $\mathbb{G}, \mathbb{D}, \mathbb{F}$, *dataset*, Z, N_e, s_b, κ);
  Gens $\leftarrow$ *Gens* $\cup \{\hat{\mathbb{G}}\}$;
return *Gens*;

misclassification of a large fraction of subjects and (2) appeared more inconspicuous (as judged by members of our team) than when training on all subjects combined. The key insight behind the algorithm is that it may be easier to find inconspicuous universal adversarial eyeglasses for similar subjects than for vastly different subjects.

Experiment Setup We tested the universal attacks against VGG143 and OF143 only, as the other DNNs were trained with too few subjects to make meaningful conclusions. To train and evaluate the generators, we selected two images for each of the subjects the DNNs were trained on—one image for training and one image for testing. To make dodging more challenging, we selected the two images that were classified correctly with the highest confidence by the two networks. Specifically, we selected images such that the product of the probabilities both DNNs assigned to the correct class was the highest among all the available images.

To explore how the number of subjects used to create the universal attacks affected performance, we varied the number of subjects with whose images we trained the adversarial generators. We averaged the success rate after repeating the process five

From \ To	VGG10	OF10
VGG10	-	63.33%
OF10	10.00%	-

From \ To	VGG143	OF143
VGG143	-	88.33%
OF143	11.67%	-

Table 5.7: Transferability of dodging in the digital domain. Each table shows how likely it is for a generator used for dodging against one network (rows) to succeed against another network (columns).

times (each time selecting a random set of subjects for training). When using ≥ 50 subjects for the universal attacks, we used Alg. 5 and set the cluster size to 10.

Additionally, we explored how the number of adversarial eyeglasses affected the success of the attack. We did so by generating 100 eyeglasses from each trained generator or set of generators and identifying the subsets (of varying size) that led the largest fraction of images in the test set to be misclassified. Finding the optimal subsets is NP-hard, and so we used an algorithm that provides a $(1 - \frac{1}{e})$-approximation of the optimal success rate [148].

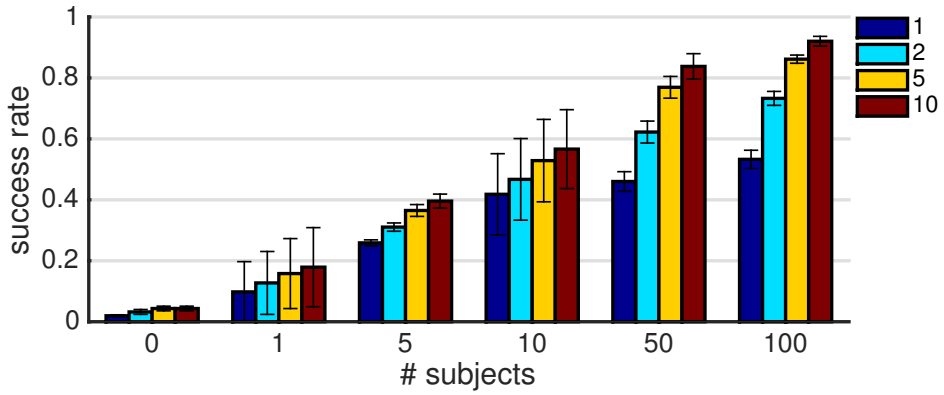
Experiment Results Fig. 5.6 summarizes the results. Universal attacks are indeed possible: generators trained to achieve dodging using a subset of subjects produced eyeglasses that led to dodging when added to images of subjects not used in training. The effectiveness of dodging depends chiefly on the number of subjects used in training and, secondarily, the number of eyeglasses generated. In particular, training a generator (set) on 100 subjects and using it to create 10 eyeglasses was sufficient to allow 92% of remaining subjects to dodge against VGG143 and 94% of remaining subjects to dodge against OF143. Even training on five subjects and generating five eyeglasses was sufficient to allow more than 50% of the remaining users to dodge against either network. OF143 was particularly more susceptible to universal attacks than VGG143 when a small number of subjects was used for training, likely due to its overall lower accuracy.

5.4.4 Transferability of Dodging Attacks

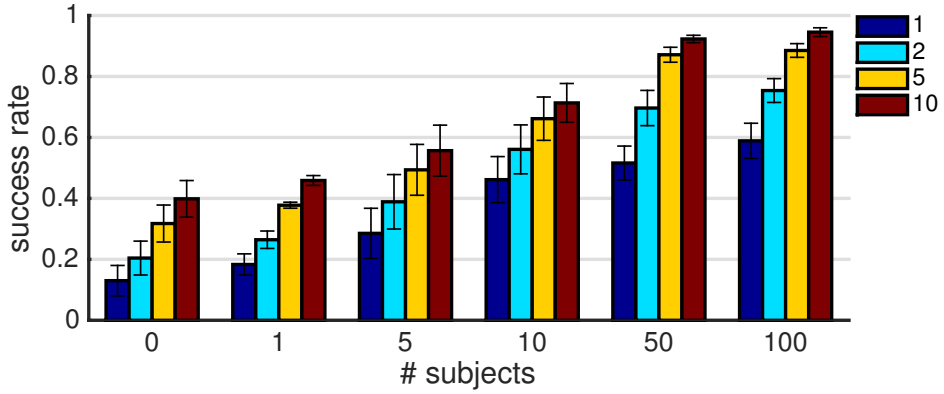
Although this is not an explicit goal of our attacks, attackers with access to one DNN but not another may attempt to rely on transferability to dodge against the second DNN. In this section, we explore whether dodging against DNNs of one architecture leads to successful dodging against DNNs of a different architecture.

Using the data from Sec. 5.4.1, we first tested whether dodging in the digital environment successfully transferred between architectures (see Table 5.7). We found that attacks against the OpenFace architecture successfully fooled the VGG architecture in only a limited number of attempts (10–12%). In contrast, dodging against VGG led to successful dodging against OpenFace in at least 63% of attempts.

Universal attacks seemed to transfer between architectures with similar success. Using attacks created with 100 subjects and 10 eyeglasses from Sec. 5.4.3, we found that 82% ($\pm$3% standard deviation) of attacks transferred from VGG143 to OF143, and 26% ($\pm$4% standard deviation) transferred in the other direction.



(a) Universal dodging against VGG143.



(b) Universal dodging against OF143.

Figure 5.6: Universal dodging against VGG143 and OF143. The x-axis shows the number of subjects used to train the adversarial generators. When the number of subjects is zero, a non-adversarial generator was used. The y-axis shows the mean fraction of images misclassified (i.e., the dodging success rate). The whiskers on the bars show the standard deviation of the success rate, computed by repeating each experiment five times, each time with a different set of randomly picked subjects. The color of the bars denotes the number of eyeglasses used, as shown in the legend. We evaluated each attack using one, two, five, or ten eyeglasses. For example, the rightmost bar in (b) indicates that an AGN trained with images of 100 subjects will generate eyeglasses such that 10 pairs of eyeglasses will allow approximately 94% of subjects to evade recognition. For ≤ 10 subjects, Alg. 4 was used to create the attacks. For 50 and 100 subjects, Alg. 5 was used.

The transferability of dodging attacks in the physical environment between architectures followed a similar trend (see Table 5.8). Successful attacks transferred less successfully from the OpenFace networks to the VGG networks (20–28%) than in the other direction (44–52%).

5.4.5 A Study to Measure Inconspicuousness

Methodology To evaluate inconspicuousness of eyeglasses generated by AGNs we carried out an online user study. Participants were told that we were developing an

From \ To	VGG10	OF10
VGG10	-	43.84%
OF10	27.77%	-

From \ To	VGG143	OF143
VGG143	-	51.78%
OF143	19.86%	-

Table 5.8: Transferability of dodging in the physical domain. We classified the frames from the physically realized attacks using DNNs different from the ones for which the attacks were crafted. Each table shows how likely it is for frames that successfully dodged against one network (rows) to succeed against another network (columns).

algorithm for designing eyeglass patterns, shown a set of eyeglasses, and asked to label each pair as either algorithmically generated or real. Each participant saw 15 "real" and 15 attack eyeglasses in random order. All eyeglasses were the same shape and varied only in their coloring. The "real" eyeglasses were ones used for pretraining the AGNs (see Sec. 5.3.1). The attack eyeglasses were generated using either AGNs, the *CCS16* attack, or the $\widehat{CCS16}$ attack.

Neither "real" nor attack eyeglasses shown to participants were photo-realistically or three-dimensionally rendered. So, we consider attack glasses to have been inconspicuous to participants not if they were uniformly rated as real (which even "real" glasses were not, particularly when attack glasses were inconspicuous), but rather if the rate at which participants deemed them as real does not differ significantly regardless of whether they are "real" eyeglasses or attack eyeglasses.

Given two sets of eyeglasses (e.g., a set of attack glasses and a set of "real" glasses), we tested whether one is more inconspicuous via the χ^2 test of independence [134], and conservatively corrected for multiple comparisons using the Bonferroni correction. We compared the magnitude of differences using the odds-ratio measure: the odds of eyeglasses in the first group being marked as real divided by the odds of eyeglasses in the second group being marked as real. The higher (resp. lower) the odds ratios are from 1, the higher (resp. lower) was the likelihood that eyeglasses from the first group were selected as real compared to eyeglasses from the second group.

We recruited 301 participants in the U.S. through the Prolific crowdsourcing service.[4] Their ages ranged from 18 to 73, with a median of 29. 51% of participants specified being female and 48% male (1% chose other or did not answer). Our study took 3 minutes to complete on average and participants were compensated $1.50. The study design was approved by Carnegie Mellon University's ethics review board.

Results Table 5.9 and Fig. 5.7 show comparisons between various groups of eyeglasses, as well as the percentage of time participants marked different eyeglasses as real. "Real" eyeglasses were more realistic than AGN-generated ones (×1.71 odds ratio). This is expected, given the additional objectives that attack eyeglasses are required to achieve. However, AGNs were superior to other attacks. Both for digital and physical attacks, eyeglasses created by AGNs were more realistic than those created by the *CCS16* attack (×2.19 and ×1.59 odds ratio, respectively). Even limiting the max-norm of the perturbations did not help—AGNs generated eyeglasses that were more likely to be selected as real than the $\widehat{CCS16}$ attack (×2.24 odds ratio).

[4] `https://prolific.ac`

Comparison (group 1 vs. group 2)		*Odds ratio*	*p-value*
Real (61%)	All AGNs (47%)	1.71	<0.01
AGNs digital (49%)	*CCS16* digital (31%)	2.19	<0.01
AGNs digital (49%)	$\widehat{CCS16}$ digital (30%)	2.24	<0.01
AGNs physical (45%)	*CCS16* physical (34%)	1.59	<0.01
AGNs:			
digital (49%)	physical (45%)	1.19	0.26
digital (49%)	digital with detector (43%)	1.28	0.02
digital dodging (52%)	universal dodging (38%)	1.80	<0.01

Table 5.9: Relative realism of selected sets of eyeglasses. For each two sets compared, we report in parentheses the fraction of eyeglasses per set that were marked as real by study participants, the odds ratios between the groups, and the p-value of the χ^2 test of independence. E.g., odds ratio of 1.71 means that eyeglasses are $\times 1.71$ as likely to be selected as real if they are in the first set than if they are in the second.

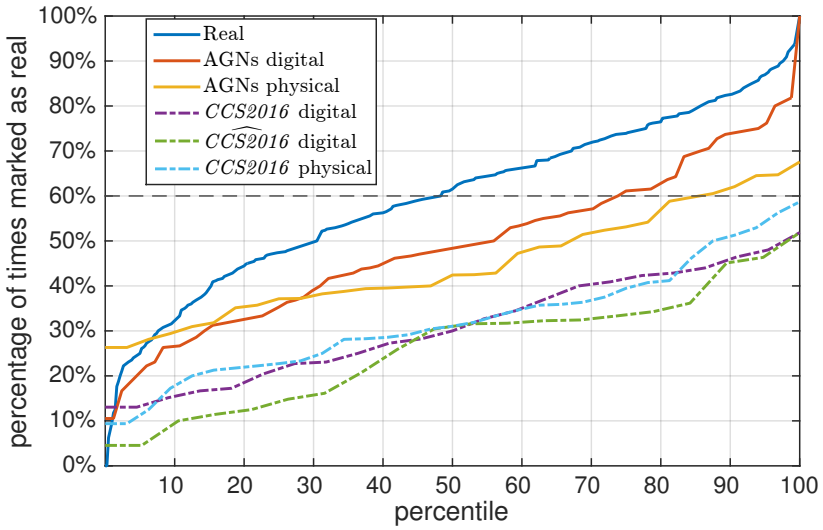


Figure 5.7: The percentage of times in which eyeglasses from different sets were marked as real. The horizontal 60% line is highlighted to mark that the top half of "real" eyeglasses were marked as real at least 60% of the time.

Perhaps most indicative of inconspicuousness in practice is that many AGN-generated eyeglasses were as realistic as "real" eyeglasses. The most inconspicuous 26% of eyeglasses emitted by AGNs for digital-environment attacks were on average deemed as real as the most inconspicuous 50% of "real" eyeglasses; in each case participants marked these eyeglasses as real >60% of the time. Physical attacks led to less inconspicuous eyeglasses; however, the 14% most inconspicuous were still marked as real at least 60% of the time (i.e., as real as the top 50% of "real" eyeglasses).

Other results match intuition—the more difficult the attack, the bigger the impact on conspicuousness. Digital attack glasses that do not try to fool a detector are less

conspicuous than ones that fool a detector (×1.28 odds ratio), and individual dodging is less conspicuous than universal dodging (×1.80 odds ratio). Digital attack glasses had higher odds of being selected as real than physical attack glasses (×1.19 odds ratio), but the differences were not statistically significant (p-*value*=0.26).

5.4.6 AGNs Against Digit Recognition

We next show that AGNs can be used in domains besides face recognition. Specifically, we use AGNs to fool a state-of-the-art DNN for recognizing digits, trained on the MNIST dataset [116], which contains 70,000 28×28-pixel images of digits.

Experiment Setup First, we trained a DNN for digit recognition using the architecture and training code of Carlini and Wagner [31]. We trained the DNN on 55,000 digits and used 5,000 for validation during training time. The trained DNN achieved 99.48% accuracy on the test set of 10,000 digits. Next, we pretrained 10 GANs to generate digits, one for each digit. Each generator was trained to map inputs randomly sampled from $[-1, 1]^{25}$ to 28x28-pixel images of digits. We again used the Deep Convolutional GAN architecture [164]. Starting from the pretrained GANs, we trained AGNs using a variant of Alg. 4 to produce generators that emit images of digits that simultaneously fool the discriminator to be real and are misclassified by the digit-recognition DNN.

Unlike prior attacks, which typically attempted to minimally perturb specific benign inputs to cause misclassification (e.g., [31, 55, 66, 157, 161, 221]), the attack we propose does not assume that a benign input is provided, nor does it attempt to produce an attack image minimally different from a benign image. Hence, a comparison with prior attacks would not be meaningful.

Experiment Results The AGNs were able to output arbitrarily many adversarial examples that appear comprehensible to human observers, but are misclassified by the digit-recognition DNN (examples are shown in Fig. 5.8). As a test, we generated 5,004 adversarial examples that all get misclassified by the digit-recognition DNN. The adversarial examples were produced by first generating 600,000 images using the adversarial generators (60,000 per generator). Out of all samples, the ones that were misclassified by the DNN (8.34% of samples) were kept. Out of these, only the digits that were likely to be comprehensible by humans were kept: the automatic filtering process to identify these involved computing the product of the discriminator's output (i.e., how realistic the images were deemed by the discriminator) and the probability assigned by the digit-recognition DNN to the correct class, and keeping the 10% of digits with the highest product.

Differently from traditional attacks on digit recognition (e.g., [31]), these attack images are not explicitly designed for minimal deviation from specific benign inputs; rather, their advantage is that they can be substantially different (e.g., in Euclidean distance) from training images. We measured the diversity of images by computing the mean Euclidean distance between pairs of digits of the same type; for attack images, the mean distance was 8.34, while for the training set it was 9.25.

A potential way AGNs can be useful in this domain is adversarial training. For instance, by augmenting the training set with the 5,004 samples, one can extend it

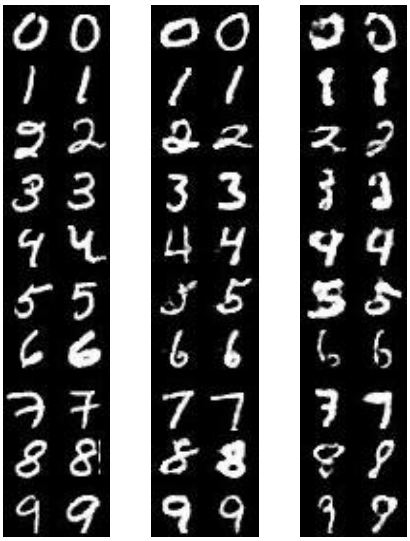


Figure 5.8: An illustration of attacks generated via AGNs. Left: A random sample of digits from MNIST. Middle: Digits generated by the pretrained generator. Right: Digits generated via AGNs that are misclassified by the digit-recognition DNN.

by almost 10%. This approach can also be useful for visualizing inputs that would be misclassified by a DNN, but are otherwise not available in the training or testing sets.

5.5 Discussion and Conclusion

In this chapter we contributed a methodology that we call *adversarial generative nets* (AGNs) to generate adversarial examples to fool DNN-based classifiers while meeting additional objectives. We focused on objectives imposed by the need to physically realize artifacts that, when captured in an image, result in misclassification of the image. Using the physical realization of eyeglass frames to fool face recognition as our driving example, we demonstrated the use of AGNs to improve robustness to changes in imaging conditions (lighting, angle, etc.) and even to specific defenses; inconspicuousness to human onlookers; and scalability in terms of the number of adversarial objects (eyeglasses) needed to fool DNNs in different contexts. AGNs generated adversarial examples that improved upon earlier work (including the work presented in Chap. 3) in all of these dimensions, and did so using a general methodology.

Our work highlights a number of features of AGNs. They are flexible in their ability to accommodate a range of objectives, including ones that elude precise specification, such as inconspicuousness. In principle, given an objective that can be described through a set of examples, AGNs can be trained to emit adversarial examples that satisfy this objective. Additionally, AGNs are general in being applicable to various domains, which we demonstrated by training AGNs to fool classifiers for face and (handwritten) digit recognition. We expect that they would generalize to other applications, as well. For example, one may consider using AGNs to fool DNNs for street-sign recognition by training the generator to emit adversarial examples that resemble street-sign images collected from the internet.

One advantage of AGNs over other attacks (e.g., [66, 206]) is that they can generate multiple, diverse, adversarial examples. Such diverse adversarial examples can be useful for evaluating the robustness of models or learning to defend against attacks (e.g., by incorporating them in adversarial training [113]).

Chapter 6

Mitigating Adversarial Examples via Ensembles of Topologically Manipulated Classifiers

6.1 Introduction

As discussed in Sec. 2.3, researchers have proposed several methods to mitigate the risks of adversarial examples. For example, adversarial training, augments the training data with correctly labeled adversarial examples (e.g., [92, 113, 129, 188]). The resulting models are often more robust in the face of attacks than models trained via standard methods. However, while defenses are constantly improving, they are still far from perfect. Relative to standard models, defenses often reduce the accuracy on benign samples. For example, methods to detect the presence of attacks sometimes erroneously detect benign inputs as adversarial (e.g., [126, 127]). Moreover, defenses often fail to mitigate a large fraction of adversarial examples that are produced by strong attacks (e.g., [10]).

Inspired by n-version programming, this chapter proposes a new defense, termed n-ML, that improves upon the state of the art in its ability to detect adversarial inputs and correctly classify benign ones. Similarly to other ensemble classifiers [128, 204, 225], an n-ML ensemble outputs the majority vote if more than a threshold number of DNNs agree; otherwise the input is deemed adversarial. The key innovation in this work is a novel method, *topological manipulation*, to train DNNs to achieve high accuracy on benign samples while simultaneously classifying adversarial examples *according to specifications that are drawn at random before training*. Because every DNN in an ensemble is trained to classify adversarial examples differently than the other DNNs, n-ML is able to detect adversarial examples because they cause disagreement between the DNNs' votes.

We evaluated n-ML using four datasets (MNIST [116], CIFAR10 [109], GTSRB [201]), and VGG [159] and against L_∞-, L_2-, and AGN-based attacks in black-, grey-, and white-box settings. Our findings indicate that n-ML can effectively mitigate adversarial examples while achieving high benign accuracy. For example, for CIFAR10 in the black-box setting, n-ML achieved 94.50% benign accuracy (vs. 95.38% for the

best standard DNN) while preventing all adversarial examples with L_∞-norm perturbation magnitudes of $\frac{8}{255}$ created by *PGD*—the best known attack algorithm for imperceptible L_∞ attacks [129]. In comparison, the state-of-the-art defense (specifically, Adv_{PGD} [129, 188]) achieved 87.24% benign accuracy while being evaded by 14.02% of the adversarial examples. n-ML is also faster than most defenses that we compared against. Specifically, even the slowest variant of n-ML is $\times$45.72 to $\times$199.46 faster at making inferences than other defenses for detecting the presence of attacks (particularly, *LID* [127] and *NIC* [126]).

Our contributions can be summarized as follows:

- We propose *topology manipulation*, a novel method to train DNNs to classify adversarial examples according to specifications that are selected at training time, while also achieving high benign accuracy.
- Using topologically manipulated DNNs, we construct (n-ML) ensembles to defend against adversarial examples.
- Our experiments using two perturbation types and four datasets in black-, grey-, and white-box settings show that n-ML is an effective and efficient defense. n-ML roughly retains the benign accuracies of state-of-the-art DNNs, while providing more resilience to attacks than the best defenses known to date, and making inferences faster than most.

We next present the technical details behind n-ML and topology manipulation (Sec. 6.2). Thereafter, we describe the experiments that we conduct and their results (Sec. 6.3 and Sec. 6.4). We close the chapter with a discussion (Sec. 6.5) and a conclusion (Sec. 6.6).

6.2 Technical Approach

Here we detail our methodology. We first briefly review the threat model. Subsequently, we present a novel technique, termed *topology manipulation*, which serves as a cornerstone for training DNNs that are used as part of the n-ML defense. Last, we describe how to construct an n-ML defense via an ensemble of topologically manipulated DNNs.

6.2.1 Threat Model

Our proposed defense aims to mitigate attacks in all settings: *black-* *grey-*, and *white-box*. In the black-box setting, the attacker attempts a non-interactive attack by transferring adversarial examples from standard surrogate models to evade classification. In the grey-box setting, the attacker is aware of the use of the defense and attempts to transfer adversarial examples that are produced against a surrogate *defended* model. In the white-box setting, the attacker can adapt gradient-based white-box attacks (e.g., *PGD*) to evade the classifier and the defense combined.[1] We do not consider interactive

[1]For clarity, we distinguish the classifier from the defense. However, in certain cases (e.g., for n-ML or adversarial training) they are inherently inseparable.

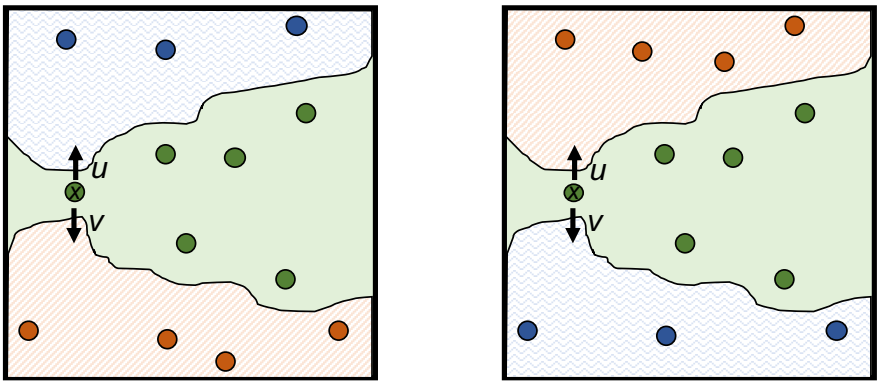


Figure 6.1: An illustration of topology manipulation. Left: In a standard DNN, perturbing the benign sample x in the direction of u leads to misclassification as blue (zigzag pattern), while perturbing it in the direction of v leads to misclassification as red (diagonal stripes). Right: In the topologically manipulated DNN, direction u leads to misclassification as red, while v leads to misclassification as blue. The benign samples (including x) are correctly classified in both cases (i.e., high benign accuracy).

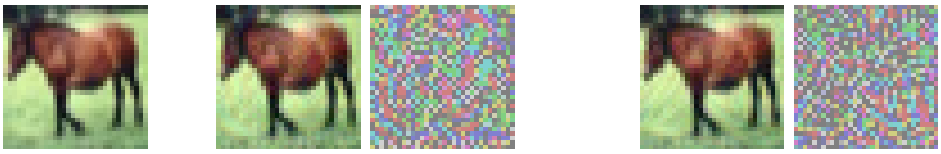
attacks that query models in a black-box setting (e.g., [23, 26, 35, 85]). These attacks are generally weaker than the white-box attacks that we do consider.

6.2.2 Topologically Manipulating DNNs

The main building block of the n-ML defense is a *topologically manipulated DNN*—a DNN that is manipulated at training time to achieve certain topological properties with respect to adversarial examples. Specifically, a topologically manipulated DNN is trained to satisfy two objectives: *1)* obtaining high classification accuracy on benign inputs; and *2)* misclassifying adversarial inputs following a certain specification. The first objective is important for constructing a well-performing DNN to solve the classification task at hand. The second objective aims to change the adversarial directions of the DNN such that an adversarial perturbation that would normally lead a benign input to be misclassified as class c_t by a regularly trained DNN would likely lead to misclassification as class $\hat{c}_t$ ($\neq c_t$) by the topologically manipulated DNN. Fig. 6.1 illustrates the idea of manipulating the topology of a DNN via an abstract example, while Fig. 6.2 gives a concrete example.

To train a topologically manipulated DNN, two datasets are used. The first dataset, D, is a standard dataset. It contains pairs (x, c_x) of benign samples, x, and their true classes, c_x. The second dataset, $\tilde{D}$, contains adversarial examples. Specifically, it consists of pairs $(\tilde{x}, c_t)$ of targeted adversarial examples, $\tilde{x}$, and the target classes, c_t. These adversarial examples are produced against reference DNNs that are trained in a standard manner (e.g., to decrease cross-entropy loss, $Loss_{ce}$). Samples in D are used to train the DNNs to satisfy the first objective (i.e., achieving high benign accuracy). Samples in $\tilde{D}$, on the other hand, are used to topologically manipulate the adversarial directions of DNNs.

More specifically, to specify the topology of the trained DNN, we use a derangement (i.e., a permutation with no fixed points), d, that is drawn at random over the



(a) **(b)** **(c)**

Figure 6.2: A concrete example of topology manipulation. The original image of a horse (a) is adversarially perturbed to be misclassified as a bird (b) and as a ship (c) by standard DNNs. The perturbations, which are limited to L_∞-norm of $\frac{8}{255}$, are shown after multiplying $\times 10$. We trained a topologically manipulated DNN to misclassify (b) as a ship and (c) as a bird, while classifying the original image correctly.

number of classes, m. This derangement specifies that an adversarial example $\tilde{x}$ in $\tilde{D}$ that targets class c_t should be misclassified as $d[c_t]$ ($\neq c_t$) by the topologically manipulated DNN. For example, for ten classes (i.e., $m = 10$), the derangement may look like $d = [1, 6, 7, 0, 2, 8, 9, 3, 5, 4]$. This derangement specifies that adversarial examples targeting class 0 should be misclassified as class $d[0] = 1$, ones targeting class 1 should be misclassified as class $d[1] = 6$, and so on. For m classes, the number of derangements the we can draw from is known as the subfactorial (denoted by $!m$), and is defined recursively as $!m = (m-1)(!(m-1)+!(m-2))$, where $!2 = 1$ and $!1 = 0$. The subfactorial grows almost as quickly as the factorial $m!$ (i.e., the number of permutations over a group of size m).

We specify the topology using derangements rather than permutations that may have fixed points because if d contained fixed points, there would exist a class c_t such that $d[c_t] = c_t$. In such case, the DNN would be trained to misclassify adversarial examples that target c_t into c_t, which would not inhibit an adversary targeting c_t. Such behavior is undesirable.

We use the standard $Loss_{ce}$ to train topologically manipulated DNNs. Formally, the training process minimizes:

$$\frac{1}{|D|} \sum_{(x,c_x)\in D} Loss_{ce}(x, c_x) + \frac{\lambda}{|\tilde{D}|} \sum_{(\tilde{x},c_t)\in \tilde{D}} Loss_{ce}(\tilde{x}, d[c_t]) \tag{6.1}$$

While minimizing the leftmost term increases the benign accuracy (as is usual in standard training processes), minimizing the rightmost term manipulates the topology of the DNN (i.e., forces the DNN to misclassify $\tilde{x}$ as $d[c_t]$ instead of as c_t). The parameter λ is a positive real number that balances the two objectives. We tune it via a hyperparameter search.

Although topologically manipulated DNNs aim to satisfy multiple objectives, it is important to point out that training them does not require significantly more time than for standard DNNs. For training, one first needs to create the dataset $\tilde{D}$ that contains the adversarial examples. This needs to be done only once, as a preprocessing phase. Once $\tilde{D}$ is created, training a topologically manipulated DNN takes the same amount of time as training a standard DNN.

6.2.3 n-ML: An Ensemble-Based Defense

As previously mentioned, n-ML is inspired by n-version programming. While n independent, or diversified, programs are used in an n-version programming defense, an n-ML defense contains an ensemble of n ($\geq$2) topologically manipulated DNNs. As explained above, all the DNNs in the n-ML ensemble are trained to behave identically for benign inputs (i.e., to classify them correctly), while each DNN is trained to follow a different specification for adversarial examples. This opens an opportunity to *1)* classify benign inputs accurately; and *2)* detect adversarial examples.

In particular, to classify an input x using an n-ML ensemble, we compute the output of all the DNNs in the ensemble on x. Then, if the number of DNNs that agree on a class is above or equal to a threshold τ, the input is classified to the majority class. Otherwise, the n-ML ensemble would abstain from classification and the input would be marked as adversarial. Formally, denoting the individual DNNs' classification results by the multiset $C = \{\mathbb{F}_i(x) | 1 \leq i \leq n\}$, the n-ML classification function, $\mathbb{F}$, is defined as:

$$\mathbb{F}(x) = \begin{cases} majority(C), & \text{if } |\{c \in C | c = majority(C)\}| \geq \tau \\ abstain, & \text{otherwise} \end{cases}$$

Of course, increasing the threshold increases the likelihood of detecting adversarial examples (e.g., an adversarial example is less likely to be misclassified as the same target class c_t by all the n DNNs than by $n-1$ DNNs). In other words, increasing τ decreases attacks' success rates. At the same time, increasing the threshold harms the benign accuracy (e.g., the likelihood of n DNNs to emit c_x is lower than the likelihood of $n-1$ DNNs to do so). In practice, we set τ to a value $\geq \lceil \frac{n+1}{2} \rceil$, to avoid ambiguity when computing the majority vote, and $\leq n$, as the benign accuracy is 0 for $\tau > n$.

Similarly to n-version programming, where the defense becomes more effective when the constituent programs are more independent and diverse [13, 34, 46], an n-ML defense is more effective at detecting adversarial examples when the DNNs are more independent. Specifically, if two DNNs i and j ($i \neq j$) are trained with derangements d_i and d_j, respectively, and we are not careful enough, there might exist a class c_t such that $d_i[c_t] = d_j[c_t]$. If so, the two DNNs are likely to classify adversarial examples targeting c_t in the same manner, thus reducing the defense's likelihood to detect attacks. To avoid such undesirable cases, we train the n-ML DNNs (simultaneously or sequentially) while attempting to avoid pairs of derangements that map classes in the same manner to the greatest extent possible. More concretely, if n is lower than the number of classes m, then we draw n derangements that disagree on all indices (i.e., $\forall i \neq j$, $\forall c_t$, $d_i[c_t] \neq d_j[c_t]$). Otherwise, we split the DNNs to groups of $m-1$ (or smaller) DNNs, and for each group we draw derangements that disagree on all indices. For a group of n DNNs, $n < m$, we can draw $\prod_{i=1}^{n}!(m-i+1)$ derangements such that every pair of derangements disagrees on all indices.

6.3 Evaluation Against L_p Attacks

In line with prior work (e.g., [126, 127, 129]), in this section we describe how we evaluated n-ML's capacity to defend against L_∞ (mainly) and L_2 (as a proof concept)

attacks. We initially present the datasets and the standard DNN architectures that we used. Then we describe how we trained individual topologically manipulated DNNs to construct n-ML ensembles, and the extent to which they met the training objectives. We close the section with experiments to evaluate the n-ML defense in various settings against L_∞ and L_2. We ran our experiments with Keras [96] and TensorFlow [1].

6.3.1 Datasets

We used three popular datasets to evaluate n-ML and other defenses against the L_p attacks: MNIST [116], CIFAR10 [109], and GTSRB [201]. MNIST is a dataset of 28×28 pixel images of digits (i.e., ten classes). It contains 70,000 images in total, with 60,000 images intended for training and 10,000 intended for testing. We set aside 5,000 images from the training set for validation. CIFAR10 is a dataset of 32×32-pixel images of ten classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset contains 50,000 images for training and 10,000 for testing. We set aside 5,000 images from the training set for validation. Last, GTSRB is a dataset containing traffic signs of 43 classes. The dataset contains 39,209 training images and 12,630 test images. We used 1,960 images that we set aside from the training set for validation. Images in GTSRB vary in size between 15×15 and 250×250. Following prior work [119], we resized the images to 48×48.

The three datasets have different properties that made them especially suitable for evaluating n-ML. MNIST relatively has a few classes, thus limiting the set of derangements that we could use for topology manipulation for high n values. At the same time, standard DNNs achieve high classification accuracy on MNIST ($\sim$99% accuracies are common), hence increasing the likelihood that DNNs in the ensemble would agree on the correct class for benign inputs. CIFAR10 also relatively has a few classes. However, differently from MNIST, even the best performing DNNs do not surpass $\sim$95% classification accuracy on CIFAR10. Consequently, the likelihood that a large number n of DNNs in an ensemble would achieve consensus may be low (e.g., an ensemble consisting of $n = 5$ DNNs with 95% accuracy each that incur independent errors could have benign accuracy as low as 77% if we require all DNNs to agree). In contrast, GTSRB contains a relatively high number of classes, and standard DNNs often achieve high classification accuracies on this dataset (98%–99% accuracies are common). As a result, there is a large space from which we could draw derangements for topology manipulation, and we had expected high benign accuracies from n-ML ensembles.

6.3.2 Standard DNNs

The DNNs that we used were based on standard architectures. We constructed the DNNs either exactly the same way as prior work or reputable public projects (e.g., by the Keras team [97]) or by modifying prior DNNs via a standard technique. In particular, we modified certain DNNs following the work of Springenberg et al. [198], who found that it is possible to construct simple, yet highly performing, DNNs by removing pooling layers (e.g., max- and average-pooling) and increasing the strides of convolutional operations.

Dataset	#	*Architecture*	*Acc.*
MNIST	1	Convolutional DNN [31]	99.42%
	2	Convolutional DNN [129]	99.28%
	3	Convolutional DNN [31] w/o pooling [198]	99.20%
	4	Convolutional DNN [97]	99.10%
	5	Convolutional DNN [129] w/o pooling [198]	99.10%
	6	Multi-layer perceptron [98]	98.56%
CIFAR10	1	Wide-ResNet-22-8 [230]	95.38%
	2	Wide-ResNet-28-10 [230]	95.18%
	3	Wide-ResNet-16-10 [230]	95.06%
	4	Wide-ResNet-28-8 [230]	94.88%
	5	Wide-ResNet-22-10 [230]	94.78%
	6	Wide-ResNet-16-8 [230]	94.78%
GTSRB	1	Convolutional DNN [119]	99.46%
	2	Same as 1, but w/o first branch [119]	99.56%
	3	Same as 1, but w/o pooling [198]	99.11%
	4	Same as 1, but w/o second branch [119]	99.08%
	5	Convolutional DNN [208]	99.00%
	6	Convolutional DNN [187]	98.07%

Table 6.1: The DNN architectures that we used for the different datasets. The DNNs' accuracies on the test sets of the corresponding datasets (after standard training) are reported to the right.

For each dataset, we trained six DNNs of different architectures—a sufficient number of DNNs to allow us to evaluate n-ML and perform attacks via transferability from surrogate ensembles [123] while factoring out the effect of architectures (see Sec. 6.3.3 and Sec. 6.3.4). The MNIST DNNs were trained for 20 epochs using the Adadelta optimizer with standard parameters and a batch size of 128 [97, 231]. The CIFAR10 DNNs were trained for 200 epochs with data augmentation (e.g., image rotation and flipping) and training hyperparameters set identically to prior work [230]. The GTSRB DNNs were trained with the Adam optimizer [99], with training hyperparameters and augmentation following prior work [119, 187, 208]. Table 6.1 reports the architectures and performance of the DNNs. In all cases, the DNNs achieved comparable performance to prior work.

6.3.3 Evaluating Individual Topologically Manipulated DNNs

Now we describe how we trained individual topologically manipulated DNNs and report on their performance. Later, in Sec. 6.3.4, we report on the performance of the n-ML ensembles.

Training When training topologically manipulated DNNs, we aimed to minimize the loss described in Eqn. 6.1. To this end, we modified the training procedures that we used to train the standard DNNs in three ways:

1. We extended each batch of benign inputs with the same number of adversarial samples $(\tilde{x}, c_t) \in \tilde{D}$ and specified that $\tilde{x}$ should be classified as $d[c_t]$.

2. In certain cases, we slightly increased the number of training epochs to improve the performance of the DNNs.

3. We avoided data augmentation for GTSRB, as we found it to harm the accuracy of (topologically manipulated) DNNs on benign inputs.

To set λ (the parameter that balances the DNN's benign accuracy and the success of topology manipulation, see Eqn. 6.1), we performed a hyperparameter search. We experimented with values in $\{0.1, 0.5, 1, 2, 10\}$ to find the best trade-off between the n-ML ensembles' benign accuracy and their ability to mitigate attacks. We found that $\lambda = 2$ achieved the highest accuracies at low attack success-rates.

To train the best-performing n-ML ensemble, one should select the best performing DNN architectures to train topologically manipulated DNNs. However, since the goal of this chapter is to evaluate a defense, we aimed to give the attacker the advantage to assess the resilience of the defense in a worst-case scenario (e.g., so that the attacker could use the better held-out DNNs as surrogates in transferability-based attacks). Therefore, we selected the DNN architectures with the lower benign accuracy to train topologically manipulated DNNs. More specifically, for each dataset, we trained n-ML ensembles by selecting round robin from the architectures shown in rows 4–6 from Table 6.1.

Constructing a dataset of adversarial examples, $\tilde{D}$, is a key part of training topologically manipulated DNNs. As we aimed to defend against attacks with bounded L_∞-norms, we used the corresponding *PGD* attack to produce adversarial examples: For each training sample x of class c_x, we produced $m-1$ adversarial examples, one targeting every class $c_t \neq c_x$. Moreover, we produced adversarial examples with perturbations of different magnitudes to construct n-ML ensembles that could resist attacks of varied strengths. For MNIST, where attacks of magnitudes $\epsilon \leq 0.3$ are typically considered [129], we used $\epsilon \in \{0.1, 0.2, 0.3, 0.4\}$. For CIFAR10 and GTSRB, where attacks of magnitudes $\epsilon \leq \frac{8}{255}$ are typically considered [129, 154], we used $\epsilon \in \{\frac{2}{255}, \frac{4}{255}, \frac{6}{255}, \frac{8}{255}\}$. (Thus, in total, $|\tilde{D}| = 4 \times (m-1) \times |D|$.) We ran *PGD* for 40 iterations, since prior work found that his leads to successful attacks [129, 188]. Additionally, to avoid overfitting to the standard DNNs that were used for training, we used state-of-the-art techniques to enhance the transferability of the adversarial examples, both by making the examples invariant to spatial transformations [51, 224] and by producing them against an ensemble of models [123, 209]—three standard DNNs of architectures 4–6.

For each dataset, we trained a total of 18 topologically manipulated DNNs. Depending on the setting, we use a different subset of the DNNs to construct n-ML ensembles (see Sec. 6.3.4). The DNNs were split into two sets of nine DNNs each ($< m$ for MNIST and CIFAR10), such that the derangements of every pair of DNNs in the same set disagreed on all indices.

Results Each topologically manipulated DNN was trained with two objectives in mind: classifying benign inputs correctly (i.e., high benign accuracy) and classifying adversarial examples as specified by the derangement drawn at training time. Here

Standard DNNs		
Dataset	*Acc.*	*TSR*
MNIST	99.30%±0.09%	43.05%±7.97%
CIFAR10	95.21%±0.13%	98.57%±0.66%
GTSRB	99.38%±0.19%	20.17%±1.48%

Topologically manipulated DNNs					
Dataset	*Acc.*	*TSR*	*TSR h/o*	*MSR*	*MSR h/o*
MNIST	98.66%±0.42%	<0.01%	6.82%±2.45%	99.98%±0.02%	53.23%±14.94%
CIFAR10	92.93%±0.39%	<0.01%	<0.01%	99.98%±0.02%	99.99%±0.01%
GTSRB	96.99%±1.45%	1.20%±0.27%	1.35%±0.27%	52.86%±9.05%	48.26%±4.68%

Table 6.2: The performance of topologically manipulated DNNs compared to standard DNNs. For standard DNNs (top), we report the average and standard deviation of the (benign) accuracy and the targeting success rate (*TSR*). *TSR* is the rate at which the DNN emited the target class on a transferred adversarial example. For topologically manipulated DNNs (bottom), we report the average and standard deviation of the accuracy, the *TSR*, and the manipulation success rate (*MSR*). *MSR* is the rate at which adversarial examples were classified as specified by the derangements drawn at training time. *TSR* and *MSR* are reported for adversarial examples produced against the same DNNs used during training or ones produced against held-out (*h/o*) DNNs.

we report on the extent to which the DNNs we trained met these objectives. Note that these DNNs were not meant to be used individually, but instead in the ensembles evaluated in Sec. 6.3.4.

To measure the benign accuracy, we classified the original (benign) samples from the test sets of datasets using the 18 topologically manipulated DNNs as well as the (better-performing) standard DNNs that we held out from training topologically manipulated DNNs (i.e., architectures 1–3). Table 6.2 reports the average and standard deviation of the benign accuracy. Compared to the standard DNNs, the topologically manipulated ones had only slightly lower accuracy (0.64%–2.39% average decrease in accuracy). Hence, we can conclude that topologically manipulated DNNs were accurate.

Next, we measured the extent to which topology manipulation is successful. To this end, we computed adversarial examples for the DNNs used to train topologically manipulated DNNs (i.e., architectures 4–6) or DNNs held out from training (i.e., architectures 1–3). Again, we used *PGD* and techniques to enhance transferability to compute the adversarial examples. As in prior work, we set $\epsilon = 0.3$ for MNIST and $\epsilon = \frac{8}{255}$ for CIFAR10 and GTSRB, and we ran *PGD* for 40 iterations. For each benign sample, x, we created $m - 1$ corresponding adversarial examples, one targeting every class $c_t \neq c_x$. To reduce the computational load, we use a random subset of benign samples from the test sets: 1024 samples for MNIST and 512 samples for the other datasets.

For constructing robust n-ML ensembles, the topologically manipulated DNNs should classify adversarial examples as specified during training, or, at least, differently than the adversary anticipates. We estimated the former via the *manipulation*

success rate (*MSR*)—the rate at which adversarial examples were classified as specified by the derangements drawn at training time—while we estimated the latter via the *targeting success rate* (*TSR*)—the rate at which adversarial examples succeeded at being misclassified as the target class. A successfully trained topologically manipulated DNN should obtain a high *MSR* and a low *TSR*.

Table 6.2 presents the average and standard deviation of *TSR*s and *MSR*s for topologically manipulated DNNs, as well as the *TSR*s for standard DNNs. One can immediately see that targeting was much less likely to succeed for a topologically manipulated DNN (average $TSR \leq 6.82\%$) than for a standard DNN (average $TSR \geq 20.17\%$, and as high as 98.57%). In fact, across all datasets and regardless of whether the adversarial examples were computed against held-out DNNs, the likelihood of targeting to succeed for standard DNNs was $\times 6.31$ or higher than for topologically manipulated DNNs. This confirms that the adversarial directions of topologically manipulated DNNs were vastly different than those of standard DNNs. Furthermore, as reflected in the *MSR* results, we found that topologically manipulated DNNs were likely to classify adversarial examples as specified by their corresponding derangements. Across all datasets, and regardless of whether the adversarial examples were computed against held-out DNNs or not, the average likelihood of topologically manipulated DNNs to classify adversarial examples according to specification was $\geq 48.26\%$. For example, an average of 99.99% of the adversarial examples produced against the held-out DNNs of CIFAR10 were classified according to specification by the topologically manipulated DNNs.

In summary, the results indicate that the topologically manipulated DNNs satisfied their objectives to a large extent: they accurately classified benign inputs, and their topology with respect to adversarial directions was different than that of standard DNNs, as they often classified adversarial examples according to the specification that was selected at training time.

6.3.4 Evaluating n-ML Ensembles

Now we describe our evaluation of n-ML ensembles. We begin by describing the experiment setup and reporting the benign accuracy of ensembles of various sizes and thresholds. We then present on experiments to evaluate n-ML ensembles in various settings and compare n-ML against other defenses. We finish with an evaluation of the time overhead incurred when deploying n-ML for inference.

6.3.4.1 Setup

The n-ML ensembles that we constructed were composed of the topologically manipulated DNNs described in the previous section. Particularly, we constructed ensembles containing five (5-ML), nine (9-ML), or 18 (18-ML) DNNs, as we found ensembles of certain sizes to be more suitable than others at balancing benign accuracy, security, and the time it takes to make inferences in different settings. For the number of variants $n \leq 9$, we selected DNNs whose derangements disagreed in all indices for the ensembles. When $n = 18$, we selected all the DNNs. Note that since GTSRB contains a large number of classes ($m = 43$), we could train 18 DNNs with derangements that disagreed on all indices. However, we avoided doing so to save compute cycles, as

the DNNs that we trained performed well despite having derangements that agreed on certain indices.

Other Defenses We compared n-ML with three state-of-the-art defenses: Adv_{PGD} [129, 188], *LID* [127], and *NIC* [126].

Adv_{PGD} and *LID* use adversarial examples at training time; we set the magnitude of the L_∞-norm perturbations to $\epsilon = 0.3$ to produce adversarial examples for MNIST, and to $\epsilon = \frac{8}{255}$ for CIFAR10 and GTSRB, as these are typical attack magnitudes that defenses attempt to prevent (e.g., [129, 154]).

For Adv_{PGD}, we implemented and used the *free adversarial training* method of Shafahi et al. [188], which adversarially trains DNNs in the same amount of time as standard training. We used Adv_{PGD} to train four defended DNNs for each dataset—one to be used by the defender, and three to be used for transferring attacks in the grey-box setting (see below). To give the defense an advantage, we used the best performing architecture for the defender's DNN (architecture 1 from Table 6.1), and the least performing architectures for the attacker's DNNs (architectures 4–6). For CIFAR10, as the DNN that we obtained after training did not perform as well as prior works', we use the adversarially trained DNN released by Madry et al. [129] as the defender's DNN.

For training *LID* detectors, we used the implementation that was published by the authors [127]. As described in Sec. 2.3, *LID* detectors compute *LID* statistics for intermediate representations of inputs and feeds the statistics to a logistic regression classifier to detect adversarial examples. The logistic regression classifier is trained using *LID* statistics of benign samples, adversarial examples, and noisy variants of benign samples (created by adding non-adversarial Gaussian noise). We tuned the amount of noise for best performance (i.e., highest benign accuracy and detection rate of adversarial examples). For CIFAR10 and GTSRB, we trained *LID* detectors for DNNs of architecture 1. For MNIST, we trained a *LID* detector for the DNN architecture that was used in the original work—architecture 4.

Using code that we obtained directly from the authors, we trained two *NIC* detectors per dataset—one to be used by the defender (in all settings), and one by the attacker in the grey-box setting. The defender's detectors were trained for DNNs of the same architectures as for *LID*. For the attacker, we trained detectors for DNNs of architectures 1, 4, and 2, for MNIST, CIFAR10, and GTSRB, respectively. We selected the attackers' DNN architectures arbitrarily, and expect that other architectures would perform roughly the same. The *oc-SVM*s that we trained for *NIC* had Radial Basis Functions (RBF) as kernels, since these were found to perform best for detection [126].

Attack Methodology We evaluated n-ML and other defenses against untargeted attacks, as they are easier to attain from the point of view of attackers, and more challenging to defend against from the point of view of the defender. For Adv_{PGD}, *LID*, and *NIC*, we used typical *PGD* untargeted attacks with various adaptations depending on the setting. Typical untargeted attacks, however, are unlikely to evade n-ML ensembles, as if the target is not specified by the attack then each DNN in the ensemble may classify the resulting adversarial example differently, thus detecting the presence of an attack. To address this, we used a more powerful attack, similarly to Carlini and Wagner [31]. The attack builds on targeted *PGD* to generate adversarial

examples targeting all possible incorrect classes (i.e., $m-1$ in total) and checks if any of these adversarial examples is misclassified by a large number of DNNs in the ensemble ($\geq \tau$), and so is not detected as adversarial by the n-ML ensemble. Because targeting every possible class increases the computational load of attacks, we used random subsets of test sets to produce adversarial examples against n-ML. In particular, we used 1,000 samples for MNIST and 512 samples for CIFAR10 and GTSRB. The magnitude of L_∞-norm of perturbations that we considered were $\epsilon \leq 0.3$ for MNIST and $\epsilon \leq \frac{8}{255}$ for CIFAR10 and GTSRB, and we ran attacks for 40 iterations. Again, we used techniques to attack ensembles and enhance the transferability of attacks [51, 123, 209, 224].

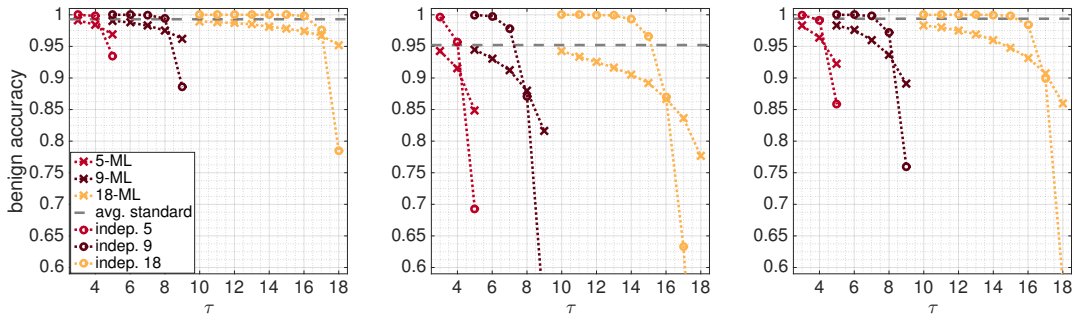
When directly attacking a DNN defended with *LID*, we simply produced adversarial examples that were misclassified with high confidence against the DNN while ignoring the defense. This approach was motivated by prior work, which found that high confidence adversarial examples mislead *LID* with high likelihood [10]. When attacking a DNN defended by *NIC*, we created a new DNN by combining the logits of the original DNN and those of the classifiers built on top of every intermediate layer. We found that forcing the original DNN and the intermediate classifiers to (mis)classify adversarial examples in the same manner often led the *oc-SVM*s to misclassify adversarial examples as benign.

Metrics In the context of adversarial examples, a defense's success is measured in its ability to prevent adversarial examples, while maintaining high benign accuracy (e.g., close to that of a standard classifier). The benign accuracy is the rate at which benign inputs are classified correctly and not detected as adversarial. In contrast, the success rate of attacks is indicative of the defense's ability to prevent adversarial examples (high success rate indicates a weak defense, and vice versa). For untargeted attacks, the success rate can be measured by the rate at which adversarial examples are not detected and are classified to a class other than the true class. Note that Adv_{PGD} is a method for robust classification, as opposed to detection, and so adversarially trained DNNs always output an estimation of the most likely class (i.e., abstaining from classifying an input that is suspected to be adversarial is not an option).

We tuned the defenses at inference time to compute different tradeoffs on the above metrics. In the case of n-ML, we computed the benign accuracy and attacks' success rates for threshold values $\lceil \frac{n+1}{2} \rceil \leq \tau \leq n$. For *LID*, we evaluated the metrics for different thresholds $\in [0, 1]$ on the logistic regression's probability estimates that inputs are adversarial. *NIC* emits scores in arbitrary ranges—the higher the score the more likely the input to be adversarial. We computed accuracy and success rate tradeoffs of *NIC* for thresholds between the minimum and maximum values emitted for benign samples and adversarial examples combined. In all cases, both the benign accuracy and attacks' success rates decreased as we increased the thresholds. Adv_{PGD} results in a single model that cannot be tuned at inference time. We report the single operating point that it achieved.

6.3.4.2 Results

We now present the results of our evaluations, in terms of benign accuracy; resistance to adversarial examples in the black-, grey-, and white-box settings; and overhead to classification performance.



(a) MNIST **(b)** CIFAR10 **(c)** GTSRB

Figure 6.3: The benign accuracy of n-ML ensembles of different sizes as we varied the thresholds. For reference, we show the average accuracy of a single standard DNN (avg. standard), as well as the accuracy of hypothetical ensembles whose constituent DNNs are assumed to have independent errors and the average accuracy of topologically manipulated DNNs (indep. n). The dotted lines connecting the markers were added to help visualize trends, but do not correspond to actual operating points.

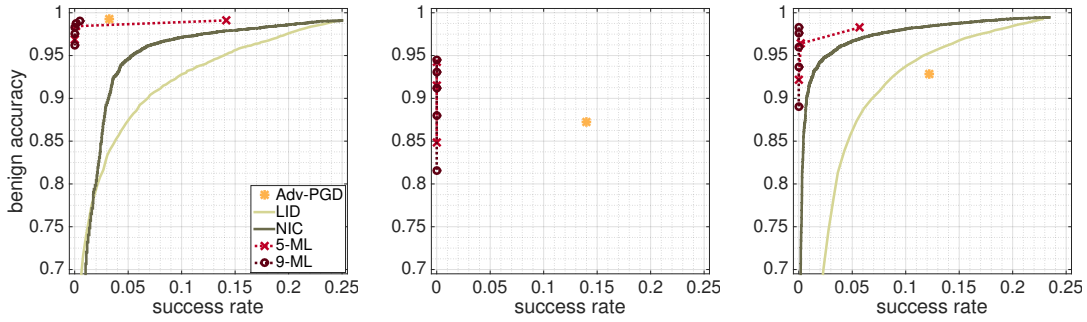
Benign Accuracy We first report on the benign accuracy of n-ML ensembles. In particular we were interested in finding how different was the accuracy of ensembles from (single) standard DNNs. Ideally, it is desirable to maintain accuracy that is as close to that of standard training as possible.

Fig. 6.3 compares the n-ML ensembles' accuracy with standard DNNs, as well as with hypothetical ensembles whose DNN members have the average accuracy of topologically manipulated DNNs and independent errors. For low thresholds, it can be seen that the accuracy of n-ML was close to the average accuracy of standard benign DNNs. As we increased the thresholds, the accuracy decreased. Nonetheless, it did not decrease as dramatically as for ensembles composed from DNNs with independent error. For example, the accuracy of a hypothetical ensemble containing five independent DNNs each with an accuracy of 92.93% (the average accuracy of topologically manipulated DNNs on CIFAR10) is 69.31% when $\tau = 5$ (i.e., we require all DNNs to agree). In comparison, 5-ML achieved 84.82% benign accuracy for the same threshold on CIFAR10.

We can conclude that the n-ML ensembles were almost as accurate as standard models for low thresholds, and that they did not suffer from dramatic accuracy loss as thresholds were increased.

Black-Box Attacks In the black-box setting, as the attacker is unaware of the use of defenses and has no access to the classifiers, we used non-interactive transferability-based attacks to transfer adversarial examples produced against standard surrogate models. For n-ML and Adv_{PGD}, we used a strong attack by transferring adversarial examples produced against the standard DNNs held-out from training the defenses. For *LID* and *NIC* we found that transferring adversarial examples produced against the least accurate standard DNNs (architecture 6) was sufficient to evade classification with high success rates.

Fig. 6.4 summarizes the results for the attacks with the highest magnitudes. For n-ML, we report the performance of 5-ML and 9-ML, which we found to perform well



(a) MNIST **(b)** CIFAR10 **(c)** GTSRB

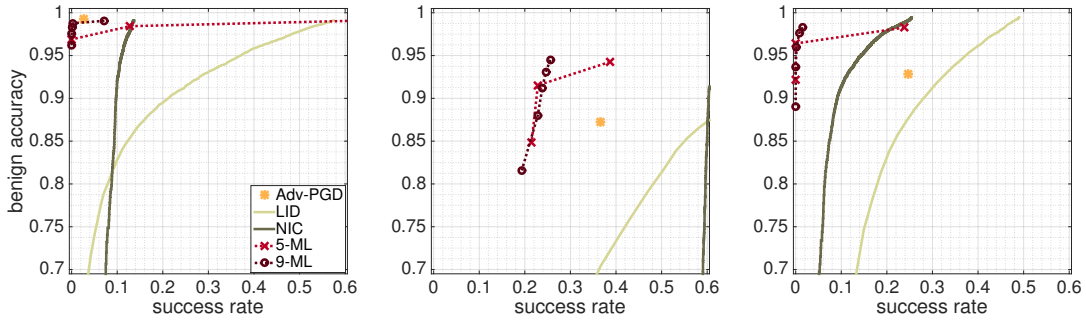
Figure 6.4: Comparison of defenses' performance in the *black-box setting*. The L_∞-norm of perturbations was set to $\epsilon = 0.3$ for MNIST and $\epsilon = \frac{8}{255}$ for CIFAR10 and GTSRB. Due to poor performance, the *LID* and *NIC* curves were left out from the CIFAR10 plot after zooming in. The dotted lines connecting the n-ML markers were added to help visualize trends, but do not correspond to actual operating points.

in the black-box setting (i.e., they achieved high accuracy while mitigating attacks). It can be seen that n-ML outperformed other defenses across all datasets. For example, for CIFAR10, 9-ML was able to achieve 94.50% benign accuracy at 0.00% attack success-rate. In contrast, the second best defense, Adv_{PGD}, achieved 87.24% accuracy at 14.02% attack success-rate.

Additional experiments that we performed demonstrated that n-ML in the black-box setting performed roughly the same as in Fig. 6.4 when *1)* different perturbation magnitudes were used ($\epsilon \in \{0.05, 0.1, \ldots, 0.4\}$ for MNIST and $\epsilon \in \{\frac{1}{255}, \frac{2}{255}, \ldots, \frac{8}{255}\}$ for CIFAR10 and GTSRB); *2)* individual standard DNNs were used as surrogates to produce adversarial examples; and *3)* the same DNNs used to train topologically manipulated DNNs were used as surrogates.

Grey-Box Attacks In the grey-box setting (where attacker are assumed to be aware of the deployment of defenses, but have no visibility to the parameters of the classifier and the defense), we attempted to transfer adversarial examples produced against surrogate defended classifiers. For n-ML, we evaluated 5-ML and 9-ML in the grey-box setting. As surrogates, we used n-ML ensembles of the same sizes and architectures to produce adversarial examples. The derangements used for training the DNNs in the surrogate ensembles were picked independently from the defender's ensembles (i.e., the derangements could agree with the defender's derangements on certain indices). For Adv_{PGD}, we used three adversarially trained DNNs different than the defender's DNNs as surrogates. For *NIC* we used standard DNNs and corresponding detectors (different than the defender's, see above for training details) as surrogates. For *LID*, we simply produced adversarial examples that were misclassified with high confidence against undefended standard DNNs of architecture 2 (these were more architecturally similar to the defenders' DNNs than the surrogates used in the black-box setting).

Fig. 6.5 presents the performance of the defenses against the attacks with the highest magnitudes. Again, we found n-ML to achieve favorable performance over other defenses. In the case of GTSRB, for instance, 9-ML could achieve 98.30% benign ac-



(a) MNIST **(b)** CIFAR10 **(c)** GTSRB

Figure 6.5: Comparison of defenses' performance in the *grey-box setting*. The L_∞-norm of perturbations was set to $\epsilon = 0.3$ for MNIST and $\epsilon = \frac{8}{255}$ for CIFAR10 and GTSRB. The dotted lines connecting the n-ML markers were added to help visualize trends, but do not correspond to actual operating points.

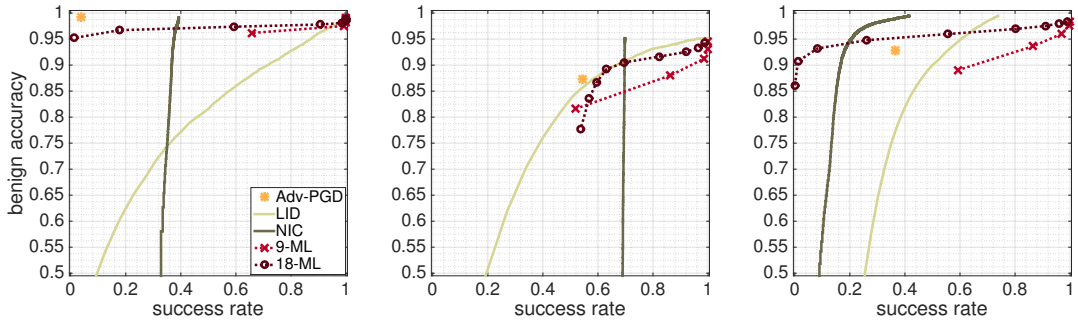
curacy at 1.56% attack success-rate. None of the other defenses was able to achieve a similar accuracy while preventing ≥98.44% of the attacks for GTSRB.

Additional experiments that we performed showed that n-ML maintained roughly the same performance as we varied the number of DNNs in the attacker's surrogate ensembles ($n \in \{1, 5, 9\}$) and the attacks' magnitudes ($\epsilon \in \{0.05, 0.1, \ldots, 0.4\}$ for MNIST and $\epsilon \in \{\frac{1}{255}, \frac{3}{255}, \frac{5}{255}, \frac{7}{255}\}$ for CIFAR10 and GTSRB).

White-Box Attacks Now we turn our attention to the white-box setting, where attackers are assumed to have complete access to classifiers' and defenses' parameters. In this setting, we leveraged the attacker's knowledge of the classifiers' and defenses' parameters to directly optimize the adversarial examples against them.

Fig. 6.6 shows the results. One can see that, depending on the dataset, n-ML outperformed other defenses, or achieved comparable performance to the leading defenses. For GTSRB, n-ML significantly outperformed other defenses: 18-ML achieved a benign accuracy of 86.01%–93.19% at attack success-rates ≤8.20%. No other defense achieved comparable benign accuracy for such low attack success-rates. We hypothesize that n-ML was particularly successful for GTSRB, since the dataset contains a relatively large number of classes, and so there was a large space from which derangements for topology manipulation could be drawn. The choice of the leading defense for MNIST and CIFAR10 is less clear (some defenses achieved slightly higher benign accuracy, while others were slightly better at preventing attacks), and depends on the need to balance benign accuracy and resilience to attacks at deployment time. For example, 18-ML was slightly better at preventing attacks against MNIST than Adv_{PGD} (1.30% vs. 3.70% attack success-rate), but Adv_{PGD} achieved slightly higher accuracy for the same 18-ML operating point (99.25% vs. 95.22%).

L_2-Norm Attacks The previous experiments showed that n-ML ensembles could resist L_∞-based attacks in various settings. We performed a preliminary exploration using MNIST to assess whether n-ML could also prevent L_2-based attacks. Specifically, we trained 18 topologically manipulated DNNs to construct n-ML ensembles. The training process was the same as before, except that we projected adversarial per-



(a) MNIST **(b)** CIFAR10 **(c)** GTSRB

Figure 6.6: Comparison of defenses' performance in the *white-box setting*. The L_∞-norm of perturbations were set to $\epsilon = 0.3$ for MNIST and $\epsilon = \frac{8}{255}$ for CIFAR10 and GTSRB. The dotted lines connecting the n-ML markers were added to help visualize trends, but do not correspond to actual operating points.

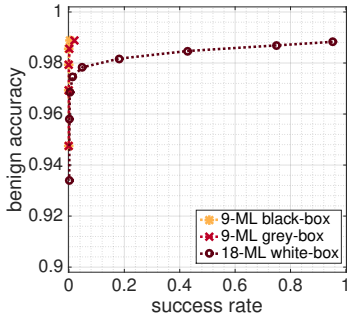


Figure 6.7: Performance of MNIST n-ML ensembles against L_2-norm *PGD* attacks with $\epsilon = 3$.

turbations to the L_2 balls around benign samples when performing *PGD* to produce adversarial examples for $\tilde{D}$. We created adversarial examples with $\epsilon \in \{0.5, 1, 2, 3\}$, as we aimed to prevent adversarial examples with $\epsilon \leq 3$, following prior work [129]. The resulting topologically manipulated DNNs were accurate (an average accuracy of 98.39%±0.55%).

Using the models that we trained, we constructed n-ML ensembles of different sizes and evaluated attacks in black-, grey-, and white-box settings. The evaluation was exactly the same as before, except that we use L_2-based *PGD* with $\epsilon = 3$ to produce adversarial examples. Fig. 6.7 summarizes the results for 9-ML in black- and grey-box settings, and for 18-ML in a white-box setting. It can be seen that n-ML was effective at preventing L_2-norm attacks while maintaining high benign accuracy. For example, 9-ML was able to achieve 98.56% accuracy at ∼0% success rates for black- and grey-box attacks, and 18-ML was able to achieve 97.46% accuracy at a success rate of 1.40% for white-box attacks.

Overhead Of course, as n-ML requires us to run several DNNs for inference instead of only one, using n-ML comes at the cost of increased inference time at deployment. Now we show that the overhead is relatively small, especially compared to *LID* and *NIC*.

Dataset	*Standard* / Adv_{PGD}	*5-ML*	*9-ML*	*18-ML*	*LID*	*NIC*
MNIST	0.15ms	×1.93	×2.80	× 4.73	×943.47	× 601.07
CIFAR10	3.72ms	×3.57	×6.07	×12.07	×551.88	× 581.51
GTSRB	0.68ms	×5.35	×8.26	×16.41	×852.04	×1457.26

Table 6.3: Defenses' overhead at inference time. The second column reports the average inference time in milliseconds for batches containing 32 samples for a single (standard or adversarially trained) DNN. The columns to its right report the overhead of defenses with respect to using a single DNN for inference.

To measure the inference time, we sampled 1024 test samples from every dataset and classified them in batches of 32 using the defended classifiers. We used 32 because it is common to use for inspecting the behavior of DNNs [71], but the trends in the time estimates remained the same with other batch sizes. We ran the measurements on a machine equipped with 64GB of memory and 2.50GHz Intel Xeon CPU using a single NVIDIA Tesla P100 GPU.

The results are shown in Table 6.3. Adv_{PGD} did not incur time overhead at inference due to producing a single DNN to be used for inference. Compared to using a single DNN, inference time with n-ML increased with n. At the same time the increase was often sublinear in n. For example, for 18-ML, the inference time increased ×4.73 for MNIST, ×12.07 for CIFAR10, and ×16.41 for GTSRB. Moreover, the increase was significantly less dramatic than for *LID* (×551.88–943.47 increase) and *NIC* (×581.51–1,457.26 increase). There, we observed that the main bottlenecks were computing the *LID* statistics for *LID*, and classification with *oc-SVM*s for *NIC*.

6.4 Evaluation Against AGNs

This section reports on an evaluation of n-ML's capability to defend against the AGN-based attacks for fooling face-recognition systems presented in Chap. 5. Particularly, the experiments we conducted measured to which extent n-ML can maintain high benign accuracy while preventing attackers from producing adversarial eyeglasses via AGNs to dodge recognition in different settings. We first present the dataset that we used and the standard DNN architecture that we build on, then we report on the performance of the n-ML ensembles.

6.4.1 Dataset

The datasets we used in Chap. 5 (containing ∼50 face images per subject) are too small to train topologically manipulated face-recognition DNNs. Hence, we used a larger dataset for training and testing the topologically manipulated and standard DNNs. In particular, we followed the URLs published by Parkhi et al. to download the face images they used to train the original VGG DNNs [159]. Parkhi et al. used roughly 1,000 images of 2,622 subjects to train the DNNs. Unfortunately, however, because some images are no longer available online, we were only able to download a

subset of the images. Of the images we downloaded, we selected ones that pertain to five males and five females that had the most images (799–835 images). In total, we selected 8,011 images, of which we used 6,449 (∼80%) for training, 801 (∼10%) for validation, and 761 (∼10%) for testing.

6.4.2 A Standard DNN

We based the DNNs that we experimented with on the OpenFace architecture [3], one of the architectures used in Chap. 5. We used the OpenFace rather than the VGG architecture because it is roughly as accurate while being lighter and taking less time to train. We again attached a classifier on top of the original OpenFace DNN that produces discriminative features of 128 dimensions, as we did in Sec. 5.3.3 (see Fig. 5.2 for the classifier's architecture). Yet, differently than before, we did not only update the classifier's weights during training. Instead, because we used a larger dataset, we updated the weights of the whole end-to-end face-recognition DNN which includes the base DNN for feature extraction and the classifier. Following Chap. 5.3.3, we augmented the training data by attaching eyeglasses to 50% of the images of each training batch to make the DNNs robust to naïve evasion attempts.

After hyperparameter search, we found that training for 30 epochs, with a batch size of 64, and the Adam optimizer [99] led to the best performing DNN. We set the learning rate to 1×10^{-4} for the first 20 epochs, and reduced it to 1×10^{-5} thereafter. The resulting DNN achieved 98.16% accuracy on the original test set, and 97.24% accuracy when attaching non-adversarial eyeglasses to the test images.

6.4.3 Evaluating n-ML Ensembles

Now we described how we constructed the n-ML ensembles and attacked them, and present their performance.

To train topologically manipulated DNNs for the n-ML ensembles, we first constructed $\tilde{D}$, the dataset containing the adversarial examples needed for training. To do so, we used AGNs to train adversarial generators to produce eyeglasses to fool the standard DNN when attached to training images. In particular, for an image x in the training set we picked a random target class $c_t \neq c_x$, and used AGNs to train a generator to produce adversarial eyeglasses. We then used the generator to generate eyeglass frames that led the image to be misclassified as the target with high confidence (above a threshold where the FPR is below 1%). Because the AGN attack is more computationally expensive than *PGD*, we did not produce adversarial examples targeting every class $c_t \neq c_x$. Instead, we randomly picked that target class and produced ∼2 adversarial counterparts for every training image. In total, we produced 14,646 adversarial examples for training (i.e., $|\tilde{D}|$=14,646).

Using the benign training images and their adversarial counterparts, we again trained a total of 18 topologically manipulated DNNs of the same architecture as the standard DNN. For training, we set the hyperparameters to the same values as for standard training. The resulting DNNs achieved high benign accuracy—97.31%±0.36% on average, comparable to the standard DNN. In the black- and grey-box settings we used

nine DNNs whose derangements disagreed on all indices to construct n-ML ensembles for defending (9-ML), while we used all DNNs in the white-box setting (18-ML).

Following our experiments from Sec. 3.3.1 and Sec. 5.4.1, we used 30 images from the test set to measure the success rates of *untargeted digital-environment attacks* against n-ML. Attackers in the digital environment can modify images on a per-pixel level to achieve misclassification, thus making the attacks challenging to defend against. We expect physical-environment attacks to be even less successful at evading n-ML. For each image, we trained nine adversarial generators via AGNs, one generator to target every class other than the true class. Using each generator, we sampled 48 adversarial eyeglasses (to cover the majority of patterns a generator could emit Sec. 5.4.2), and tested whether any led to misclassification by the n-ML ensembles (i.e., in total we produced $9\times48=432$ variants per test image, and checked whether any was misclassified).

To evaluate attacks in the black-box setting, we attempted to transfer adversarial examples that were produced against the standard DNN as the surrogate. In the grey-box setting, we used nine held-out topologically manipulated DNNs to construct a surrogate n-ML ensemble to transfer the attacks from. To produce attacks against the ensemble, we adapted the AGNs algorithm to compute the loss with respect to all DNNs in the ensemble when training the generator, instead of computing the loss with respect to a single DNN. In particular, we defined the adapted loss to be the summation of the losses of each individual DNN (see Alg. 4). In the white-box setting, we produced attacks directly against the the n-ML18 ensemble using the adapted AGN algorithm.

Fig. 6.8 shows the benign accuracy of the n-ML models at different attack success-rates, as we varied the threshold τ. It can be seen that n-ML was able to effectively prevent attacks while maintaining high benign accuracy. In the black-box setting, for example, 9-ML was able to achieve 95.27% benign accuracy while preventing 96.67% (i.e., 29 of 30) of the attack attempts. In comparison, digital-environment black-box attacks against a single OpenFace DNN trained to recognize 10 subjects succeeded in 63.33%, even when transferred from a DNN of a different architecture (see Sec. 5.4.4). In the white-box setting, 18-ML was able to achieve 92.64% benign accuracy while preventing 46.66% (i.e., 14 of 30) of the attack attempts. In comparison, all evasion attempts against the standard DNN in the white-box setting succeeded. As expected, better tradeoffs could be achieved in the grey-box setting (e.g., 94.35% accuracy while preventing 56.67% of the attacks).

6.5 Discussion

Our experiments demonstrated the effectiveness and efficiency of n-ML in various settings. Still, there are some limitations and practical considerations to take into account when deploying n-ML. We discuss these below.

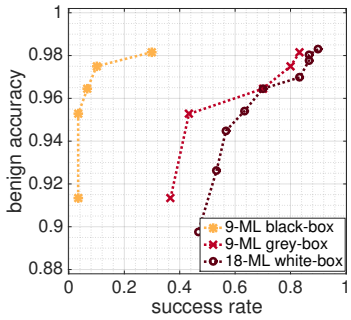


Figure 6.8: Performance of face-recognition n-ML ensembles against AGN-based attacks.

6.5.1 Limitations

Our experiments evaluated n-ML against L_∞-, L_2-, and AGN-based attacks. However, in reality, attackers can use other perturbation types to evade n-ML (e.g., by adversarially rotating images [55]). Conceptually, it should be possible to train n-ML to defend against other perturbation types. We defer the evaluation to future work.

As opposed to using a single ML algorithm for inference (e.g., one standard DNN), n-ML requires using n DNNs. As a result, more compute resources and more time are needed to make inferences with n-ML. This may make it challenging to deploy n-ML in settings where compute resources are scarce and close to real-time feedback is expected (e.g., face-recognition on mobile phones). Nonetheless, it is important to highlight that n-ML is remarkably faster at making inferences than state-of-the-art methods for detecting adversarial examples [126, 127], as our experiments showed.

Currently, perhaps the most notable weakness of n-ML is that it is limited to scenarios where the number of classes, m, is large. In cases where m is small, one cannot draw many distinct derangements to train DNNs with different topologies with which to construct n-ML ensembles. For example, when there are two classes ($m = 2$), there is only one derangement that one can use to train a topologically manipulated DNN (remember that $!2 = 1$), and so it is not possible to construct an ensemble containing $n \geq 2$ DNNs with distinct topologies. A possible solution is to find a new method that does not require derangements to topologically manipulate DNNs. We plan to pursue this direction in future work.

6.5.2 Practical Considerations

ML systems often take actions based on inferences made by ML algorithms. For example, a biometric system may give or deny access to users based on the output of a face-recognition DNN; an autonomous vehicle may change the driving direction, accelerate, stop, or slow down based on the output of a DNN for pedestrian detection; and an anti-virus program may delete or quarantine a file based on the output of an ML algorithm for malware detection. This raises the question of how should a system that uses n-ML for inference react when n-ML flags an input as adversarial.

We have a couple of suggestions for courses of action that are applicable to different settings. One possibility is to fall back to a more expensive, but less error prone,

classification mechanism. For example, if an n-ML ensemble is used for face recognition and it flags an input as adversarial, a security guard may be called to identify the person, and possibly override the output of the ensemble. This solution is viable when the time and resources to use an expensive classification mechanism are available. Another possibility is to resample the input, or classify a transformed variant of the input, to increase the confidence in the detection or to correct the classification result. For example, if an n-ML ensemble that is used for face recognition on a mobile phone detects an input as adversarial, the user may be asked to reattempt identifying herself using a new image. In this case, because the benign accuracy of n-ML is high and the attack success-rate is low, a benign user is likely to succeed at identifying, while an attacker is likely to be detected.

6.6 Conclusion

This chapter presented n-ML, a defense against adversarial examples. n-ML uses ensembles of DNNs to classify inputs by a majority vote (when a large number of DNNs agree) and to detect adversarial examples (when the DNNs disagree). To ensure that the ensembles have high accuracy on benign samples while also defending against adversarial examples, n-ML trains the DNNs using a novel technique, (*topology manipulation*), which allows to specify how adversarial examples should be classified by the DNN at inference time. Our experiments using three perturbation types (ones with bounded L_2- and L_∞-norms, and ones produced via AGNs) and four datasets (MNIST, CIFAR10, GTSRB, and VGG) in black-, grey-, and white-box settings showed that n-ML is an effective and efficient defense. In particular, n-ML roughly retained the benign accuracies of state-of-the-art DNNs, while providing more resilience to attacks than the best defenses known to date and making inferences faster than most.

Chapter 7

Summary and Future Work

7.1 Summary

While early work showed how adversarial examples that are close to benign inputs in L_p distances can evade ML algorithms, the risk they pose to ML systems in practical settings remained speculative. This thesis confirmed our speculations by showing how to systematically create adversarial examples that satisfy multiple objectives to practically mislead state-of-the-art ML algorithms in two application domains (face recognition and malware detection), even when the objectives necessary for practical evasion are hard to specify precisely. In addition, to help protect ML systems and their users, the thesis proposed a novel defense against adversarial examples.

In particular, in the face-recognition domain, we showed that attackers can create *inconspicuous* and *physically realizable* eyeglasses to dodge recognition or impersonate specific subjects when donned (Chap. 3). To create such eyeglasses, the attackers need to solve an optimization problem to ensure not only that the eyeglasses mislead a face-recognition system, but that they *1)* do so from multiple poses; *2)* have smooth transitions between neighboring pixels; and *3)* contain colors that can be physically realized by a commodity printer. Our experiments showed that our proposed attack can successfully mislead a state-of-the-art face-recognition DNN in the physical environment (e.g., $\geq$80% success rate in all dodging attempts).

In the malware-detection domain, we showed that attackers can manipulate functional code of binaries in a *functionality preserving* manner to mislead DNNs for malware detection (Chap. 4). To this end, the attacker guides binary-diversification techniques via optimization to iteratively transform binaries till evasion occurs. Our attack achieved near 100% success rates when evading three DNNs in different settings.

We further proposed the AGN framework to generate adversarial examples that satisfy multiple objectives, and particularly ones that elude precise specification, such as inconspicuousness (Chap. 5). To do so, our framework builds on the GANs framework [65] to train a generator neural network to produce artifacts that evade ML algorithms while being similar to artifacts from a target distribution. Using face-recognition as a main application domain, we showed that our proposed framework can produce inconspicuous eyeglasses (often deemed more inconspicuous than real eyeglasses by user-study participants) to fool face recognition with even higher success rates than

our first attack, and while satisfying additional objectives (e.g., evading recognition under varied lighting conditions).

Last, we proposed n-ML, a defense against adversarial examples via ensembles of diversified DNNs that is inspired by n-version programming. The key novelty of n-ML is a training method we call *topology manipulation* to train DNNs to achieve high benign accuracy while classifying adversarial examples according to a specification set at training time. At inference time, n-ML uses DNNs that are trained with different specification in an ensemble and classifies inputs by a vote. As the DNNs mostly agree on benign inputs and disagree on adversarial examples, n-ML is able to detect attacks. Our experiments with four datasets and three attacks (including AGNs), showed that n-ML roughly maintains the benign accuracy of state-of-the-art DNNs while being more resilient against attacks in black-, grey-, and white-box settings than prior defenses, and achieving lower test-time overhead that the majority of defenses we tested.

7.2 Future Work

This section points out future directions to extend or apply the work presented in this thesis.

7.2.1 Attacks in New Applications Domains

Attacks with multiple objectives akin to the ones studied in this thesis may be applicable for evaluating the security of ML algorithms in other applications domains. For example, anomaly detection algorithms are often used to protect safety-critical Industrial Control Systems from harm (e.g., a cyber attack on a water treatment can lead to poisoning drinking water [118]). To evaluate such algorithms in practical settings, one may develop attacks to produce adversarial examples that adhere to certain physical constraints (e.g., readings of a water-pressure sensor have to be in certain ranges).

It would also be interesting to extend the attacks presented in this thesis to evaluate the security of systems that use sensors other than the ones we considered. For example, in the context of face-recognition systems, we considered systems that use traditional visible-light cameras. However, certain systems use different sensors, such as infrared cameras [140], for face recognition. By extending our attacks, it would be possible to explore the effect of sensors on ML systems' security. Early work in this direction has already shown that using a variety of sensor types can enhance security [32].

We believe that the attacks that we presented in Chap. 4 (against DNNs for malware detection) can be extended to help study and improve the robustness clone-search methods (e.g., [50]) that are often used in reverse engineering for studying new malware, detecting patent infringements, or finding vulnerabilities in software. Ding et al. recently suggested the use of neural networks to map assembly code to vector representations that are similar for clones and different for non-clones [50]. Building on our attacks, we believe that attackers can manipulate the representations generated by such neural networks to make the representations of clones different (e.g., to make it difficult to study new malware), or those of non-clones similar (e.g., to support a fake patent infringement case).

7.2.2 Leveraging Simulation to Test Attacks and Defenses

The physically realizable attacks against face-recognition systems that we showed in Chap. 3 and Chap. 5 required considerable (physical) effort. As a result, it was infeasible to test a large number of attack attempts in the physical environment. To address this, future work may develop realistic simulation engines in which it would be possible to simulate how a face would look like in various locations (e.g., indoors or outdoors) and under different conditions (e.g., changing poses and illuminations) to evaluate attacks as well as defenses more efficiently and at a larger scale. Of course, to accurately estimate attacks' and defenses' performance, it would be important to model the physical world as realistically as possible. In fact, our preliminary exploration of this direction were hindered by the lack of simulation realism.

7.2.3 Leveraging Attacks to Improve Security

While the majority of the work presented in this thesis explored how systems can fail (as is often the case in computer security), it is important to ensure that the knowledge that we accumulated would help make systems and ML algorithms more secure. One (rather traditional) way to do so is by building on our knowledge of how systems can be attacked to develop defenses, as we did in Chap. 6. A novel way that we aim to leverage the attacks in to improve security is by deceiving attackers that can be faithfully modeled as ML algorithms to protect systems and users' data. For example, attackers seeking to guess users' passwords [94, 136], or ones attempting fingerprint the web traffic of anonymous communication systems' users [167, 196], can often be modeled via ML algorithms. In these settings, adversarial examples against the ML models can inform how the defender can prevent attacks.

7.2.4 Making n-ML Training More Efficient and General

While training a single topologically manipulated DNN for an n-ML ensemble takes as much time as training a standard DNN (see Sec. 6.2), training n DNNs takes $\times n$ as much. In the future, it would be useful to explore ways to reduce the amount of time that is needed for training to increase the efficiency and practicality of the defense. A possible way to do so would be to train topologically manipulated DNNs starting from a pretrained DNN that already performs well on benign inputs. Such training process may require less time to converge, thus reducing the total time needed for training n DNNs.

Additionally, to make n-ML more practical, it would be useful to generalize it for additional settings and attack types. These include *1)* developing methods to topologically manipulate large number of DNNs when the number of classes is small (i.e., finding ways to specify or manipulate topologies that do not require derangements); *2)* generalizing the notion of topology manipulation to classifiers other than DNNs that are also vulnerable against adversarial examples (e.g., SVMs [59]); and *3)* defending against multiple perturbation types at once, akin to recent work [131].

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proc. OSDI*, 2016.

[2] M. Abbasi and C. Gagné. Robustness to adversarial examples through an ensemble of specialists. In *Proc. ICLRW*, 2017.

[3] B. Amos, B. Ludwiczuk, and M. Satyanarayanan. OpenFace: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[4] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[5] H. S. Anderson and P. Roth. Ember: An open dataset for training static PE malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[6] S. E. Armoun and S. Hashemi. A general paradigm for normalizing metamorphic malwares. In *Proc. FIT*, 2012.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.

[8] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proc. SODA*, 2007.

[9] A. Athalye and N. Carlini. On the robustness of the CVPR 2018 white-box adversarial example defenses. *arXiv preprint arXiv:1804.03286*, 2018.

[10] A. Athalye, N. Carlini, and D. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. ICML*, 2018.

[11] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok. Synthesizing robust adversarial examples. In *Proc. ICML*, 2018.

[12] Autodesk. Measuring light levels. `https://goo.gl/hkBWbZ`.

[13] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[14] T. Baltrušaitis, P. Robinson, and L.-P. Morency. Openface: An open source facial behavior analysis toolkit. In *Proc. WACV*, 2016.

[15] S. Baluja and I. Fischer. Adversarial transformation networks: Learning to generate adversarial examples. In *Proc. AAAI*, 2018.

[16] B. Barak, N. Bitansky, R. Canetti, Y. T. Kalai, O. Paneth, and A. Sahai. Obfuscation for evasive functions. In *Proc. TCC*, 2014.

[17] A. D. Bethke. *Genetic Algorithms As Function Optimizers*. PhD thesis, University of Michigan, 1980.

[18] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Proc. ECML PKDD*, 2013.

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

[20] F. Biondi, M. Enescu, T. Given-Wilson, A. Legay, L. Noureddine, and V. Verma. Effective, efficient, and robust packing detection and classification. *Computers and Security*, 2018.

[21] A. J. Booker, J. Dennis Jr, P. D. Frank, D. B. Serafini, V. Torczon, and M. W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural optimization*, 17(1):1–13, 1999.

[22] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proc. COMPSTAT*, 2010.

[23] W. Brendel, J. Rauber, and M. Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *Proc. ICLR*, 2018.

[24] N. Brown and T. Sandholm. Libratus: The superhuman ai for no-limit poker. In *Proc. IJCAI*, 2017.

[25] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Proc. CAV*, 2011.

[26] T. Brunner, F. Diehl, M. T. Le, and A. Knoll. Guessing smart: Biased sampling for efficient black-box adversarial attacks. In *Proc. ICCV*, 2019.

[27] Y. Cao, C. Xiao, D. Yang, J. Fang, R. Yang, M. Liu, and B. Li. Adversarial objects against Lidar-based autonomous driving systems. *arXiv preprint arXiv:1907.05418*, 2019.

[28] CaretDashCaret. 3d printable frames from eyeglasses SVGs. `https://github.com/caretdashcaret/pince-nez`. Online; accessed 27 Oct 2019.

[29] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou. Hidden voice commands. In *Proc. USENIX Security*, 2016.

[30] N. Carlini and D. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. AISec*, 2017.

[31] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Proc. IEEE S&P*, 2017.

[32] V. Chandrasekaran, B. Tang, V. Pendyala, K. Fawaz, S. Jha, and X. Wu. Enhancing ml robustness using physical-world constraints. *arXiv preprint arXiv:1905.10900*, 2019.

[33] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *Proc. BMVC*, 2014.

[34] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. ISFTC*, 1995.

[35] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proc. AISec*, 2017.

[36] S.-T. Chen, C. Cornelius, J. Martin, and D. H. P. Chau. Shapeshifter: Robust physical adversarial attack on faster r-cnn object detector. In *Proc. ECML PKDD*, 2018.

[37] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proc. CCS*, 2018.

[38] M. Christodorescu and S. Jha. Testing malware detectors. In *Proc. ISSTA*, 2004.

[39] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. IEEE S&P*, 2005.

[40] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report, University of Wisconsin-Madison, 2005.

[41] Chronicle. Virustotal. `https://www.virustotal.com/`, 2004–. Online; accessed 27 Oct 2019.

[42] M. Cisse, Y. Adi, N. Neverova, and J. Keshet. Houdini: Fooling deep structured prediction models. In *Proc. NIPS*, 2017.

[43] J. M. Cohen, E. Rosenfeld, and J. Z. Kolter. Certified adversarial robustness via randomized smoothing. In *Proc. ICML*, 2019.

[44] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[45] S. Coull and C. Gardner. What are deep neural networks learning about malware? http://tiny.cc/FireEyeDNN, 2018. Online; accessed 27 Oct 2019.

[46] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. USENIX Security*, 2006.

[47] Cylance Inc. Cylance: Artificial intelligence based advanced threat prevention. `https://www.cylance.com/en-us/index.html`, 2019. Online; accessed 27 Oct 2019.

[48] H. Dang, Y. Huang, and E.-C. Chang. Evading classifiers by morphing in the dark. In *Proc. CCS*, 2017.

[49] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

[50] S. H. Ding, B. C. Fung, and P. Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proc. IEEE S&P*, 2019.

[51] Y. Dong, T. Pang, H. Su, and J. Zhu. Evading defenses to transferable adversarial examples by translation-invariant attacks. In *Proc. CVPR*, 2019.

[52] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proc. MHS*, 1995.

[53] S. Eberz, N. Paoletti, M. Roeschlin, M. Kwiatkowska, I. Martinovic, and A. Patané. Broken hearted: How to attack ECG biometrics. In *Proc. NDSS*, 2017.

[54] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *Proc. ICCV*, 1999.

[55] L. Engstrom, D. Tsipras, L. Schmidt, and A. Madry. A rotation and a translation suffice: Fooling CNNs with simple transformations. In *Proc. NeurIPSW*, 2017.

[56] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[57] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song. Robust physical-world attacks on machine learning models. In *Proc. CVPR*, 2018.

[58] H. Fan, Z. Cao, Y. Jiang, Q. Yin, and C. Doudou. Learning deep face representation. *arXiv preprint arXiv:1403.2802*, 2014.

[59] A. Fawzi, O. Fawzi, and P. Frossard. Analysis of classifiers' robustness to adversarial perturbations. *arXiv preprint arXiv:1502.02590*, 2015.

[60] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[61] R. Feng and B. Prabhakaran. Facilitating fashion camouflage art. In *Proc. 21st ACM International Conference on Multimedia*, pages 793–802. ACM, 2013.

[62] W. Fleshman, E. Raff, J. Sylvester, S. Forsyth, and M. McLean. Non-negative networks against adversarial attacks. *arXiv preprint arXiv:1806.06108*, 2018.

[63] J. Gilmer, R. P. Adams, I. Goodfellow, D. Andersen, and G. E. Dahl. Motivating the rules of the game for adversarial example research. *arXiv preprint arXiv:1807.06732*, 2018.

[64] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[65] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proc. NIPS*, 2014.

[66] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *Proc. ICLR*, 2015.

[67] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

[68] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *Proc. ESORICS*, 2017.

[69] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser. The Cuckoo Sandbox. `https://cuckoosandbox.org/`, 2012. Online; accessed 27 Oct 2019.

[70] C. Guo, M. Rana, M. Cisse, and L. van der Maaten. Countering adversarial images using input transformations. In *Proc. ICLR*, 2018.

[71] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. In *Proc. SOSP*, 2019.

[72] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[73] A. Harvey. CV Dazzle: Camouflage from face detection. Master's thesis, New York University, 2010. Available at: http://cvdazzle.com.

[74] R. Hassan, B. Cohanim, O. De Weck, and G. Venter. A comparison of particle swarm optimization and the genetic algorithm. In *Proc. MDO*, 2005.

[75] W. He, J. Wei, X. Chen, N. Carlini, and D. Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *Proc. WOOT*, 2017.

[76] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proc. NSPW*, 2009.

[77] Hex-Rays. IDA: About. `https://www.hex-rays.com/products/ida/`. Online; accessed 27 Oct 2019.

[78] H. Hosseini and R. Poovendran. Semantic adversarial examples. In *Proc. CVPRW*, 2018.

[79] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983*, 2017.

[80] B. Huang, Y. Wang, and W. Wang. Model-agnostic adversarial detection by random perturbations. In *Proc. IJCAI*, 2019.

[81] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[82] L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proc. AISec*, 2011.

[83] R. Huang, B. Xu, D. Schuurmans, and C. Szepesvári. Learning with a strong adversary. *CoRR, abs/1511.03034*, 2015.

[84] W. Huang and J. W. Stokes. MtNet: A multi-task neural network for dynamic malware classification. In *Proc. DIMVA*, 2016.

[85] A. Ilyas, L. Engstrom, and A. Madry. Prior convictions: Black-box adversarial attacks with bandits and priors. In *Proc. ICLR*, 2019.

[86] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proc. IWSPA*, 2018.

[87] L. Introna and H. Nissenbaum. Facial recognition technology: A survey of policy and implementation issues. Technical report, Center for Catastrophe Preparedness and Response, New York University, 2009.

[88] Itseez. OpenCV: Open Source Computer Vision. `http://opencv.org/`.

[89] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-llvm–software protection for the masses. In *Proc. IWSP*, 2015.

[90] C. Kanbak, S.-M. Moosavi-Dezfooli, and P. Frossard. Geometric robustness of deep networks: analysis and improvement. In *Proc. CVPR*, 2018.

[91] H. Kannan, A. Kurakin, and I. Goodfellow. Adversarial logit pairing. *arXiv preprint arXiv:1803.06373*, 2018.

[92] A. Kantchelian, J. Tygar, and A. D. Joseph. Evasion and hardening of tree ensemble classifiers. In *Proc. ICML*, 2016.

[93] S. Kariyappa and M. K. Qureshi. Improving adversarial robustness of ensembles with diversity training. *arXiv preprint arXiv:1901.09981*, 2019.

[94] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE S&P*, 2012.

[95] J. Kennedy, D. Batchelor, C. Robertson, M. Satran, and M. LeBLanc. PE format. `https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format`, 2019. Online; accessed 3 June 2019.

[96] Keras team. Keras: The Python deep learning library. `https://keras.io/`, 2015. Online; accessed 30 Sep 2019.

[97] Keras team. MNIST CNN. `https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py`, 2018. Online; accessed 28 Sep 2019.

[98] Keras team. MNIST MLP. `https://keras.io/examples/mnist_mlp/`, 2018. Online; accessed 28 Sep 2019.

[99] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, 2015.

[100] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. USENIX Security*, 2009.

[101] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *Proc. EUSIPCO*, 2018.

[102] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744, 2006.

[103] J. Z. Kolter and E. Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proc. ICML*, 2018.

[104] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *Proc. IEEE S&P*. IEEE, 2018.

[105] H. Koo and M. Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proc. AsiaCCS*, 2016.

[106] N. Koren. Color management and color science. `http://www.normankoren.com/color_management.html`. Online; accessed 27 Oct 2019.

[107] M. Krčál, O. Švec, M. Bálek, and O. Jašek. Deep convolutional malware classifiers can learn from raw executables and labels only. In *Proc. ICLRW*, 2018.

[108] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Adversarial examples on discrete sequences for beating whole-binary malware detection. In *Proc. NeurIPSW*, 2018.

[109] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[110] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. USENIX Security*, 2004.

[111] N. Kumar, A. C. Berg, P. N. Belhumeur, and S. K. Nayar. Attribute and simile classifiers for face verification. In *Proc. ICCV*, 2009.

[112] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. In *Proc. ICLRW*, 2017.

[113] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial machine learning at scale. In *Proc. ICLR*, 2017.

[114] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Proc. IEEE S&P*, 2014.

[115] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proc. ICML*, 2014.

[116] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`, 1998. Online; accessed 1 Oct 2019.

[117] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. Certified robustness to adversarial examples with differential privacy. In *Proc. IEEE S&P*, 2019.

[118] D. Li, D. Chen, B. Jin, L. Shi, J. Goh, and S.-K. Ng. MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks. In *Proc. ICANN*, 2019.

[119] J. Li and Z. Wang. Real-time traffic sign recognition based on efficient CNNs in the wild. *IEEE Transactions on Intelligent Transportation Systems*, 20(3):975–984, 2018.

[120] B. Liang, M. Su, W. You, W. Shi, and G. Yang. Cracking classifiers for evasion: A case study on the Google's phishing pages filter. In *Proc. WWW*, 2016.

[121] F. Liao, M. Liang, Y. Dong, T. Pang, J. Zhu, and X. Hu. Defense against adversarial attacks using high-level representation guided denoiser. In *Proc. CVPR*, 2018.

[122] X. Liu, M. Cheng, H. Zhang, and C.-J. Hsieh. Towards robust neural networks via random self-ensemble. In *Proc. ECCV*, 2018.

[123] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. In *Proc. ICLR*, 2017.

[124] D. Lowd and C. Meek. Adversarial learning. In *Proc. KDD*, 2005.

[125] P.-H. Lu, P.-Y. Chen, and C.-M. Yu. On the limitation of local intrinsic dimensionality for characterizing the subspaces of adversarial examples. *arXiv preprint arXiv:1803.09638*, 2018.

[126] S. Ma, Y. Liu, G. Tao, W.-C. Lee, and X. Zhang. NIC: Detecting adversarial samples with neural network invariant checking. In *Proc. NDSS*, 2019.

[127] X. Ma, B. Li, Y. Wang, S. M. Erfani, S. Wijewickrema, G. Schoenebeck, D. Song, M. E. Houle, and J. Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. In *Proc. ICLR*, 2018.

[128] F. Machida. N-version machine learning models for safety critical systems. In *Proc. DSN DSMLW*, 2019.

[129] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proc. ICLR*, 2018.

[130] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. In *Proc. CVPR*, 2015.

[131] P. Maini, E. Wong, and J. Z. Kolter. Adversarial robustness against the union of multiple perturbation models. *arXiv preprint arXiv:1909.04068*, 2019.

[132] T. Malzbender, D. Gelb, and H. Wolters. Polynomial texture maps. In *Proc. SIGGRAPH*, 2001.

[133] H. Massalin. Superoptimizer: A look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[134] M. L. McHugh. The chi-square test of independence. *Biochemia Medica*, 23(2):143–149, 2013.

[135] Megvii Inc. Face++. `http://www.faceplusplus.com/`.

[136] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proc. USENIX Security*, 2016.

[137] D. Meng and H. Chen. Magnet: A two-pronged defense against adversarial examples. In *Proc. CCS*, 2017.

[138] X. Meng, B. P. Miller, and S. Jha. Adversarial binaries for authorship identification. *arXiv preprint arXiv:1809.08316*, 2018.

[139] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff. On detecting adversarial perturbations. In *Proc. ICLR*, 2017.

[140] Microsoft. Windows Hello face authentication. `http://tiny.cc/MSHelloIR`, 2017. Online; accessed 27 Oct 2019.

[141] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proc. ICML*, 2018.

[142] T. Miyato, S.-i. Maeda, M. Koyama, K. Nakae, and S. Ishii. Distributional smoothing with virtual adversarial training. In *Proc. ICLR*, 2016.

[143] MobileSec. Mobilesec Android Authentication Framework. `https://github.com/mobilesec/authentication-framework-module-face`.

[144] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal adversarial perturbations. In *Proc. CVPR*, 2017.

[145] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: A simple and accurate method to fool deep neural networks. In *Proc. CVPR*, 2016.

[146] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. ACSAC*, 2007.

[147] NEC. Face recognition. `http://www.nec.com/en/global/solutions/biometrics/technologies/face_recognition.html`.

[148] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions. *Mathematical Programming*, 14(1):265–294, 1978.

[149] NEURO Technology. SentiVeillance SDK. `http://www.neurotechnology.com/sentiveillance.html`.

[150] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[151] P. S. Oliveto, T. Paixão, J. P. Heredia, D. Sudholt, and B. Trubenová. How to escape local optima in black box optimisation: When non-elitism outperforms elitism. *Algorithmica*, 80(5):1604–1633, 2018.

[152] T. Pang, C. Du, Y. Dong, and J. Zhu. Towards robust detection of adversarial examples. In *Proc. NeurIPS*, 2018.

[153] T. Pang, K. Xu, C. Du, N. Chen, and J. Zhu. Improving adversarial robustness via promoting ensemble diversity. In *Proc. ICML*, 2019.

[154] N. Papernot and P. McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

[155] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: From phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[156] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proc. AsiaCCS*, 2017.

[157] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *Proc. IEEE Euro S&P*, 2016.

[158] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. IEEE S&P*, 2012.

[159] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *Proc. BMVC*, 2015.

[160] J. Pinheiro, D. Bates, S. DebRoy, D. Sarkar, and R Core Team. *nlme: Linear and Nonlinear Mixed Effects Models*, 2015. R package version 3.1-122.

[161] O. Poursaeed, I. Katsman, B. Gao, and S. Belongie. Generative adversarial perturbations. In *Proc. CVPR*, 2018.

[162] Y. Qin, N. Carlini, I. Goodfellow, G. Cottrell, and C. Raffel. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In *Proc. ICML*, 2019.

[163] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proc. USENIX Security*, 2019.

[164] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *Proc. ICLR*, 2016.

[165] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole exe. In *Proc. AAAIW*, 2018.

[166] A. Raghunathan, J. Steinhardt, and P. S. Liang. Semidefinite relaxations for certifying robustness to adversarial examples. In *Proc. NeurIPS*, 2018.

[167] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen. Automated website fingerprinting through deep learning. In *Proc. NDSS*, 2018.

[168] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: A review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[169] M. Roberts. Virusshare. `https://virusshare.com/`, 2012. Online; accessed 18 June 2019.

[170] P. Robinette, W. Li, R. Allen, A. M. Howard, and A. R. Wagner. Overtrust of robots in emergency evacuation scenarios. In *Proc. HRI*, 2016.

[171] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[172] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *Proc. RAID*, 2018.

[173] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)*, 46(1):4, 2013.

[174] A. Rozsa, E. M. Rudd, and T. E. Boult. Adversarial diversity and hard positive generation. In *Proc. CVPRW*, 2016.

[175] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[176] S. Sabour, Y. Cao, F. Faghri, and D. J. Fleet. Adversarial manipulation of deep representations. In *Proc. ICLR*, 2016.

[177] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. In *Proc. NIPS*, 2016.

[178] H. Salman, G. Yang, J. Li, P. Zhang, H. Zhang, I. Razenshteyn, and S. Bubeck. Provably robust deep learning via adversarially trained smoothed classifiers. In *Proc. NeurIPS*, 2019. To appear.

[179] H. Salman, G. Yang, H. Zhang, C.-J. Hsieh, and P. Zhang. A convex relaxation barrier to tight robustness verification of neural networks. In *Proc. NeurIPS*, 2019. To appear.

[180] P. Samangouei, M. Kabkab, and R. Chellappa. Defense-GAN: Protecting classifiers against adversarial attacks using generative models. In *Proc. ICLR*, 2018.

[181] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proc. ASPLOS*, 2013.

[182] L. Schönherr, K. Kohls, S. Zeiler, T. Holz, and D. Kolossa. Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. In *Proc. NDSS*, 2019.

[183] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proc. CVPR*, 2015.

[184] M. Sconzo. Packer yara ruleset. `https://github.com/sooshie/packerid`, 2014. Online; accessed 18 June 2019.

[185] A. Sen, X. Zhu, L. Marshall, and R. Nowak. Should adversarial attacks use pixel p-norm? *arXiv preprint arXiv:1906.02439*, 2019.

[186] S. Sengupta, T. Chakraborti, and S. Kambhampati. MTDeep: Moving target defense to boost the security of deep neural nets against adversarial attacks. In *Proc. GameSec*, 2019.

[187] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In *Proc. IJCNN*, 2011.

[188] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein. Adversarial training for free! In *Proc. NeurIPS*, 2019. To appear.

[189] M. Sharif, L. Bauer, and M. K. Reiter. On the suitability of lp-norms for creating and preventing adversarial examples. In *Proc. CVPRW*, 2018.

[190] M. Sharif, L. Bauer, and M. K. Reiter. n-ML: Mitigating adversarial examples via ensembles of topologically manipulated classifiers. In *Proc. IEEE S&P*, 2020. Under submission.

[191] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. CCS*, 2016.

[192] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. A general framework for adversarial examples with objectives. *ACM Transactions on Privacy and Security (TOPS)*, 2019.

[193] M. Sharif, K. Lucas, L. Bauer, M. K. Reiter, and S. Shintre. Optimization-guided binary diversification to mislead neural networks for malware detection. In *Proc. NDSS*, 2020. Under submission.

[194] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[195] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[196] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proc. CCS*, 2018.

[197] C. Sitawarin, A. N. Bhagoji, A. Mosenia, P. Mittal, and M. Chiang. Rogue signs: Deceiving traffic sign recognition with malicious ads and logos. *arXiv preprint arXiv:1801.02780*, 2018.

[198] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *Proc. ICLR*, 2015.

[199] V. Srinivasan, A. Marban, K.-R. Müller, W. Samek, and S. Nakajima. Counterstrike: Defending deep learning architectures against adversarial samples by Langevin dynamics with supervised denoising autoencoder. *arXiv preprint arXiv:1805.12017*, 2018.

[200] N. Srndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. IEEE S&P*, 2014.

[201] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, 32:323–332, 2012.

[202] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007.

[203] Y. Steinbuch. JetBlue ditching boarding passes for facial recognition. *New York Post*, May 31 2017.

[204] T. Strauss, M. Hanselmann, A. Junginger, and H. Ulmer. Ensemble methods as a defense to adversarial perturbations against deep neural networks. *arXiv preprint arXiv:1709.03423*, 2017.

[205] O. Suciu, S. E. Coull, and J. Johns. Exploring adversarial examples in malware detection. In *Proc. AAAIW*, 2018.

[206] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Proc. ICLR*, 2014.

[207] P. Szor. *The Art of Computer Virus Research and Defense*. Pearson Education, 2005.

[208] L. Tian. Traffic sign recognition using CNN with learned color and spatial transformation. `http://tiny.cc/GTSRB_STCNN`, 2017. Online; accessed on 28 Sep 2019.

[209] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble adversarial training: Attacks and defenses. In *Proc. ICLR*, 2018.

[210] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. IEEE S&P*, 2015.

[211] A. Vedaldi and K. Lenc. MatConvNet – convolutional neural networks for MATLAB. In *Proc. ACMMM*, 2015.

[212] D. Vijaykeerthy, A. Suri, S. Mehta, and P. Kumaraguru. Hardening deep neural networks via adversarial model cascades. 2019.

[213] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proc. CVPR*, 2001.

[214] VirusTotal. Packer yara ruleset. `https://github.com/Yara-Rules/rules/tree/master/Packers`, 2016. Online; accessed 18 June 2019.

[215] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proc. CCS*, 2002.

[216] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *Proc. SCAM*, 2006.

[217] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *Proc. ICSE*, 2019.

[218] S. Wang, P. Wang, and D. Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *Proc. SANER*, 2016.

[219] X. Wang, S. Wang, P.-Y. Chen, Y. Wang, B. Kulis, X. Lin, and P. Chin. Protecting neural networks with hierarchical random switching: Towards better robustness-accuracy trade-off for stochastic defenses. *arXiv preprint arXiv:1908.07116*, 2019.

[220] E. Wong, F. R. Schmidt, and J. Z. Kolter. Wasserstein adversarial examples via projected sinkhorn iterations. *arXiv preprint arXiv:1902.07906*, 2019.

[221] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song. Generating adversarial examples with adversarial networks. In *Proc. IJCAI*, 2018.

[222] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, and D. Song. Spatially transformed adversarial examples. In *Proc. ICLR 2018*, 2018.

[223] C. Xie, Y. Wu, L. van der Maaten, A. Yuille, and K. He. Feature denoising for improving adversarial robustness. *arXiv preprint arXiv:1812.03411*, 2018.

[224] C. Xie, Z. Zhang, Y. Zhou, S. Bai, J. Wang, Z. Ren, and A. L. Yuille. Improving transferability of adversarial examples with input diversity. In *Proc. CVPR*, 2019.

[225] H. Xu, Z. Chen, W. Wu, Z. Jin, S.-y. Kuo, and M. Lyu. NV-DNN: Towards fault-tolerant DNN systems with N-version programming. In *Proc. DSN DSMLW*, 2019.

[226] W. Xu, D. Evans, and Y. Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *Proc. NDSS*, 2018.

[227] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proc. NDSS*, 2016.

[228] T. Yamada, S. Gohshi, and I. Echizen. Privacy Visor: Method based on light absorbing and reflecting properties for preventing face image detection. In *Proc. SMC*, 2013.

[229] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Proc. NIPS*, 2014.

[230] S. Zagoruyko and N. Komodakis. Wide residual networks. In *Proc. BMVC*, 2016.

[231] M. D. Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[232] Q. Zeng, J. Su, C. Fu, G. Kayas, L. Luo, X. Du, C. C. Tan, and J. Wu. A multiversion programming inspired approach to detecting audio adversarial examples. In *Proc. DSN*, 2019.

[233] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu. DolphinAttack: Inaudible voice commands. In *Proc. CCS*, 2017.

[234] Z. Zhao, D. Dua, and S. Singh. Generating natural adversarial examples. In *Proc. ICLR*, 2018.

[235] E. Zhou, Z. Cao, and Q. Yin. Naive-deep face recognition: Touching the limit of LFW benchmark or not? *arXiv preprint arXiv:1501.04690*, 2015.

[236] H. Zhou, W. Li, Y. Zhu, Y. Zhang, B. Yu, L. Zhang, and C. Liu. DeepBillboard: Systematic physical-world testing of autonomous driving systems. *arXiv preprint arXiv:1812.10812*, 2018.