

A PROPOSED DEFENSE FRAMEWORK AGAINST
ADVERSARIAL MACHINE LEARNING ATTACKS USING
HONEYPOTS.

by

Fadi Younis

B.Sc. Ryerson University, Toronto (ON), Canada, 2009

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2018

© Fadi Younis 2018

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public for the purpose of scholarly research only.

Abstract

A proposed Defense Framework Against Adversarial Machine Learning Attacks Using Honeypots.

Fadi Younis

Master of Science, Computer Science

Ryerson University, 2018

This is my wonderful abstract. I really like my abstract because it's so pretty. I've made sure it's less than 150 words if I'm a M.Sc. student and less than 350 words if I'm a Ph.D. student. I've not included any graphs, tables or references in it. If I really had to use symbols in my abstract I'd be sure to also explain right here what they stand for because I'm such a good student and I know all the thesis rules.

Acknowledgements

Many people have contributed to my work here at Ryerson University. First I thank my supervisor Dr. Ali Miri for guiding my research, as well as providing many helpful suggestions throughout my time here. I would like to acknowledge the Department of Computer Science for providing me funding while I did my research.

To

*To my friends, My dear family and and wonderful
colleagues, Without whom none of my success would be
possible.*

Table of Contents

1	Introduction	1
1.1	Setting	1
1.2	The Problem at a Glance	2
1.3	Motivation	3
1.4	Thesis Goals	4
1.5	Overview	5
2	Background	6
2.1	Security of Deep Learning Network	6
2.1.1	Deep Neural Networks	6
2.1.2	Adversarial Deep Learning	8
2.1.3	Deep Learning Threats	9
2.2	Adversarial Examples	10
2.2.1	Adversarial Examples Definition	10
2.2.2	Adversarial Example Properties	11
2.2.3	Adversarial Example Origins	12
2.2.4	Generating Adversarial Examples	12
2.2.5	The Adversarial Optimization Problem	15
2.2.6	Impact of Adversarial Examples on Deep Neural Networks	16
2.2.7	Combating Adversarial Examples - Defenses	17

2.3	Transferability and Black-Box Learning Systems	19
2.3.1	Adversarial Transferability	19
2.3.2	Black-Box Threat Model	21
2.3.3	Black-box Threat Model Vs. Blind Threat Model	22
2.3.4	Transferability in Black-Box Attacks	23
2.3.5	Transferability of Adversarial Examples in Black-Box Attacks . . .	24
2.3.6	Black-Box Attack Approach	24
2.3.7	Defense Strategies Against Black-Box Attacks	27
2.4	Honeypots	29
2.4.1	Concept of Honeypots	29
2.4.2	Classification of Honeypots	29
2.4.3	Honeypot Deployment Modes	31
2.4.4	Honeypot Role and Responsibilities	31
2.4.5	Honeypots Level of Interaction	33
2.4.6	Uses of Honeypots	34
2.5	Honeypot in our Solution	35
3	Related Work	37
4	Proposed Defense Approach	41
4.1	Problem Definition	42
4.2	Assumptions	42
4.3	Design Decisions	45
4.4	The Threat model	48
4.4.1	The Adversary	48
4.4.2	Attacker Capabilities	48
4.4.3	Attack Setting	49
4.4.4	Exploited Vulnerability	50

4.4.5	Attack Strategy	50
4.5	Decoy DNN Model	50
4.6	Adversarial HoneyPot Network Overview	50
4.7	Individual HoneyPot Topology	50
4.7.1	Purpose	50
4.7.2	Architecture and Topology	50
4.7.3	Training, Testing and Validation	50
4.8	Attracting The Adversary	51
4.8.1	Adversarial Tokens	51
4.8.2	Weak TCP/IP ports	51
4.8.3	Decoy Target Model	51
4.9	Detecting Malicious Behavior	51
4.10	Monitoring the Adversary	51
4.11	Launching The Attack	51
4.12	Defending Against Attack	51
4.13	Deployment	51
4.14	Scalability	51
4.15	Security	51
4.16	Significance and Novelty	51
5	Implementation of Adversarial HoneyTokens Component	54
5.1	Background	54
5.2	Project Structure	56
5.3	Architecture	56
5.4	Features	58
5.5	Functionality	59
5.6	Usage	63
5.7	External Dependencies	68

5.8	Integration	70
5.9	Benefits	71
6	Conclusion and Future Work	72
6.1	Summary	72
6.2	Discussion	72
6.3	Contributions	72
6.4	Future Work	72
6.5	Conclusion	72
7	Appendix A	73
7.1	Adversarial Honey Source Code	73
7.1.1	contentgen.go	73
7.1.2	txtemail template	76
7.1.3	hbconf.yaml	77
7.1.4	honeybits.go	82
7.2	Scenario1	94
7.2.1	attachment 1.1 - initializing the Go environment	94
7.2.2	attachment 1.2 - running and deploying the tokens	94
7.2.3	attachment 1.3 - deploying email token	95
7.2.4	attachment 1.4	95
7.2.5	attachment 1.5	95
7.2.6	attachment 1.6	96
7.2.7	attachment 1.7	96
7.2.8	attachment 1.8	97
7.2.9	attachment 1.9	98
	Bibliography	99

List of Tables

List of Figures

2.1	A classic DNN architecture. In this particular model, the DNN recognizes images of handwritten digit integers 1...9 and calculates the probability of the image being in one of the N=10 classes.	8
2.2	Adversarial Examples - Input \vec{x} (left), perturbation ε (middle), adversarial example \vec{x}^* (right)	10
2.3	An Adversarial Example is generated by modifying a legitimate input sample, in such a way which would cause to the classifier to mislabel it, where as a human wouldn't notice a difference.	11
2.4	Adversarial Regions in Classifier Decision Space	13
2.5	Jacobian Saliency Map Approach	15
2.6	Adversarial Example Optimization Problem	15
2.7	Convex vs. non-convex optimization	16
2.8	cross-technique Transferability matrix: cell (i,j)is the percentage of adversarial samples crafted to mislead a classifier learned using machine learning technique i that are misclassified by a classifier trained with technique j. .	20
2.9	A Simple Black-Box System	21
2.10	Adversarial Example Optimization Problem	22
2.11	Substitute Model Training	26
2.12	Classic HoneyPot Architecture	30
2.13	Low Interaction HoneyPot	34
2.14	High Interaction HoneyPot	35
2.15	Adversarial Example Optimization Problem	36
5.1	Adversarial HoneyToken Architecture	57
5.2	Adversarial Token Leakage	61
5.3	Dockerize Adversarial HoneyToken Application	62
5.4	Scenario1 - Luring Away Attacker from Target Model	64
5.5	Scenario 2 - discourage future attacks	68
5.6	Scenario 3 - apprehend an internal adversary	68
7.1	Scenario 1 - attachment 1.1	94
7.2	Scenario 1 - attachment 1.2	94
7.3	Scenario1 - attachment 1.3	95
7.4	Scenario 1 - attachment 1.4	95

7.5	Scenario1 - attachment 1.5	95
7.6	Scenario1 - attachment 1.6	96
7.7	Scenario1 - attachment 1.7	96
7.8	Scenario1 - attachment 1.8	97
7.9	Scenario1 - attachment 1.9	98

Chapter 1

Introduction

In this chapter, we introduce the thesis problem setting in which Adversarial Examples occur and maliciously degrade the prediction model (*Section 1.1*). Then, we provide an overview of the thesis problem subject matter (*Section 1.2*), the motivation for solving the problem follows after (*Section 1.3*). We then outline the goal(s) we hope to reach with our solution, by the end of this thesis (*Section 1.4*). Finally, we end the chapter with an outline for the remainder of this thesis, including chapter outlines and contents(*Section 1.5*).

1.1 Setting

Machine Learning, as we know it, is exploding in development and demand, with utilization in critical applications, services and domains, not confined to one area of industry but several. With each day, more applications are harnessing the suitability of Machine Learning. To meet the ever increasing demand that this forefront is witnessing, tech giants such as Amazon, Google, Uber, Netflix, Microsoft and many others are providing their Machine Learning adjuncted products and services in the form of an online cloud service, otherwise known as *Machine-Learning-As-A-Service* (MLaaS) [30]. While the need for easily accessible Machine Learning tools is becoming readily available, the desire to personally customize and build these services from the ground up is actually decreasing and less relied upon. This is because users do not wish to spend countless hours training, testing and fine-tuning their machine learning models, they simply want to readily use them. While some users still prefer to ordain and control how their models are constructed and deployed, companies have undergone the effort to hide the internal complex mechanisms from most of their indolent users, and package them in

non-transparent and easily customized services. Essentially, they provide their services in the form of a *Black-Box* [18] [26]. This opaque system container accepts some input and produces an output, but where the internal details of the model are hidden. However, like any application deployed, we cannot assume it is sited in a safe environment. There are security flaws and vulnerabilities in every man-made system and the container holding a MLaaS is no exception to the assumption. These weaknesses introduce a susceptibility to malicious attack(s), which is expected and almost always the case. Like any regular user, an adversarial attacker also does not have access to the internal components of the system, this knowledge might give a sense of security, and that the *Black-Box* system is secure. But as we will see, this sense of security is only temporary, and only the beginning of the dangers to follow.

1.2 The Problem at a Glance

As we learned in the introduction of this chapter, Machine Learning adversarial threats exist and are lurking close-by. The threat transpires when an attacker misleads and confuses the prediction model inside the cloud computing application offering the MLaaS, and allow malicious activities to go undetected [26]. This drives up the rate of false negatives, violating model integrity. These masqueraded inputs - called Adversarial Examples [18], represent one of the recent threats to cloud services providing *Machine Learning as a Service* (MLaaS). These nonlinear inputs look familiar to the inputs normally accepted by a linearly designed classifier, but only appear that way. They're maliciously crafted to exploit blind spots in the classifier boundary space, to mislead and confuse the learning mechanism in the classifier, to compromise the model integrity, post training. Most defenses aim at strengthening the classifiers discriminator function, by training it on malicious input ahead of time to make it robust. Defense methods, such as *Regularization and Adversarial Training* have proven unsuccessful. The latter method, and others like it, alone, cannot be relied upon since they do not generalize well on new adversarial inputs. This is evidently true in the case of a Black-Box (blind model) setting, where the adversary has access to only input and output labels, as mentioned. Our aim is to develop an adversarial defense framework to act as a secondary level of prevention to curb adversarial examples from corrupting the classifier, by deceiving the attacker.

1.3 Motivation

The market demand for machine learning is increasing, and with that the risk of adversarial threats has increased, also. For example, an attacker can maliciously fool an *Artificial Neural Network* (ANN) classifier into allowing malicious activities to go undetected, without direct influence on the classifier itself [23]. These masqueraded inputs - *Adversarial Examples*, represent one of the recent threats to cloud services providing Machine Learning as a Service (MLaaS). They're maliciously crafted to exploit *blind spots* in the classifier boundary space. Exploiting these blind spots can be used to mislead and confuse the learning mechanism in the classifier, post model training, for purposes of violating the integrity of the model. As a result, there has been an increased interest in defense techniques to combat them and robustify classifiers against attacks.

Our challenge here lies in constructing an adversarial defense technique, capable of deceiving an intrusive attacker and lure him away from the black-box target model. For purposes of our approach, we have decided to primarily use *Adversarial Honey-Tokens* as one of the methods to accomplish this, which acts as fictional digital *breadcrumbs* designed to lure the attacker, and made conspicuously detectable, to be discovered by the adversary. It is possible to generate a unique token for each item (or a sequence) to deceive the attacker and track his abuse. However each token must be strategically designed, generated and embedded into the system to misinform and fool the adversary.

Previous research on adversarial defense methods has aimed at strengthening the classifiers discriminator function, by training it on malicious input to make it robust. The latter defense methods, such as *Regularization and Adversarial Training* have proven unsuccessful. The latter method has been criticized because it cannot be relied upon, since it does not generalize well on new adversarial inputs [31]. This is evidently true in the case of a Black-Box (blind model) setting, where the adversary has access to only input and output labels. We believe it is necessary to develop an adversarial defense framework to act as a secondary level of protection to prevent adversarial examples from corrupting the classifier, by deceiving the attacker and prevent the attack from ever transpiring. The majority of our efforts is focused on designing a decentralized network of High-Interaction honeypots as an open target for adversaries. This network of honeypot nodes act as self-contained *sandboxes* to contain the decoy neural network, collect valuable data, and gain insight into adversarial attacks. We believe this might confound and deter adversaries from attacking the target model. Other adversarial example defenses can also benefit and utilize this framework as an appendage in their techniques. Unlike

other proposed defense models in literature, our model prevents the attacker from interacting directly with the target.

We designed our defense framework to deceive the adversary in three consecutive steps, occurring in sequential order of each other. The information collected from the attacker’s interaction with decoy model could then potentially be used to learn from the attacker, re-train and robustify the deep learning model in future training iterations. Our defense approach is motivated by trying to answer the following question *”is there a computationally cheap and possible way to fool the attacker and prevent him from learning behavior of the model, even before the attack occurs ”*. At its core, our intention is to devise a defense technique to both fool and prevent the attacker from interacting with the model.

1.4 Thesis Goals

Our thesis is as follows:

”Given a deep learning classification model $F()$, within a private black-box setting, and a intrusive attacker with an adversarial input \vec{x}^ capable of corrupting the model and misclassifying its labels; there exists a defense framework $D_{defense}$ to support existing defense systems, fool and deter the attacker’s attempts. If building such a defense framework is possible, then there exists a way to mislead and prevent the attacker from initially learning the model’s $F()$ behavior and corrupting it.”*

The purpose of this work is to investigate the thesis above and build an appropriate defense framework that implements it. Due to time and resource constraints, we limit our objectives to the following:

- Propose an adversarial defense approach, that will act as a secondary level of defense to reinforce existing adversarial defense mechanisms. It aims to: 1) preventing the attacker from correctly learning the classifier labels, and approximating the correct architecture of the Black-Box system, 2) luring the attacker away from the target model towards a decoy model, and 3) create an infeasible computational work for the adversary, with no functional use or benefit.
- Provide a detailed architecture and implementation of the *Adversarial Honey-Tokens*, their design, features, usage, deployment benefits, evaluation etc.

1.5 Overview

This thesis has 7 chapters. It is divided as follows:

- *Chapter 1 - gives a brief introduction to the problem at hand and its setting. This chapter also gives an overview of the thesis goals, and a breakdown of the outline.*
- *Chapter 2 - introduces relevant concepts and background knowledge, such as Deep Neural Networks (DNN), Adversarial Transferability, Adversarial Examples, Black-Box Systems, and Honeypots.*
- *Chapter 3 - gives a summary and critical evaluation of the related work(s) authored by other researchers on the topic of adversarial black box defenses, as well as specific work done with honeypots.*
- *Chapter 4 - outlines the design and architecture of the defense approach. It also gives insight into the approach's setup, environment and limitations. It also shows how Honeypots can be used to curb adversarial attacks from influencing and exploiting the model.*
- *Chapter 5 - provides an implementation of the Adversarial Honey-Tokens, its features, usage, deployment and benefits.*
- *Chapter 6 - details the approach set-up, adversarial environment, limitations and evaluation criteria.*
- *Chapter 7 - summarizes the thesis, gives an overview of the contribution(s) made, and suggests future research directions.*

Chapter 2

Background

The aim of this Chapter is to introduce the main problems and challenges involved in defenses against *Adversarial Examples*. We also formulate and explain important concepts so that they can be used to understand the upcoming chapters. First, we discuss the Security of Deep Neural Networks, as well as their over all security, architecture, design, and briefly mention the threats that imperil them (*section 2.1*). Then, we explore the concept of Adversarial Examples, as well as their properties, origins, process of generation, impact and known defenses against them (*section 2.2*). Next, we discuss Adversarial Transferability, Black-Box Threat Model, Transferability in Black-Box Attacks and the Black-Box Attack Approach, as well as defenses against Black-Box Attacks. Finally, We end the chapter with an in depth section on Honeypots, their classification, deployment modes, roles and responsibilities, advantages and disadvantages, levels of interaction, uses, and how honeypots fit into our solution (*section 2.3*).

2.1 Security of Deep Learning Network

In this section, we cannot start discussing attacking Deep Neural Networks without first exploring what a Deep Neural Network (DNN) actually is, and how they operates. Here, we also try to briefly discuss the threats that jeopardize the integrity of Deep Neural Networks.

2.1.1 Deep Neural Networks

According to [26], a *Deep Neural Network* (DNN) is a widely known machine learning technique that utilizes n parametric functions to model an input sample \vec{x} , where \vec{x} could be an image, text, video, etc. These structures (DNNs) are used to *teach* computers to

sovereignly learn from prior knowledge and experience without any explicit intervention or guidance from a human being. The *Deep* keyword in the title stems from the number of *deep* layers inside the underlying neural network architecture. They differ from conventional neural networks in the sense that conventional neural networks are non-*deep* or *shallow*, this means they contain only one or two hidden layer. Having a few layers limits the ability of the model learn features and solve complex problems. DNNs' can solve difficult and often complicated problems. Among the countless uses for DNNs', they can be used to build image classification systems that can identify an object from the edges, features, depth and color of that particular object. All of that information is processed in the hidden layers of the model, known as the *deep* layers, and as the number of these *deep* layers increases, so does the ability of the DNN to solve more complex tasks. For a detailed illustration of a DNN, see Figure 2.1 for details, provided to us by [26].

Each parametric function f_i represents a layer i in the DNN, where each layer i comprises a sequence of *perceptrons* (artificial neurons), modeled to form a chain sequence. Each neuron maps an input x to an output y using an *activation function* $f(\varphi)$. Each neuron is influenced by a parameterized *weight vector* represented by θ_i . The weight vectors holds the knowledge of the DNN when it comes to preparing a model F . A DNN computes and defines model a F as follows [26]:

$$F(\vec{x}) = f_n(\theta_i, f_{n-1}(\theta_{n-1}, \dots f_2(\theta_2, f_1(\theta_1, \vec{x}))))$$

The *training phase* in the DNN occurs when the model F learns the values for the layer weights $\theta_F = \{\theta_1, \theta_2, \dots, \theta_n\}$. The model is given a large of set of input/output represented by (\vec{x}, \vec{y}) and works by adjusting the weights and reducing (minimizing) the difference (error) between the predicted value of $F(\vec{x})$ and what the output \vec{y} should be (usually done using back propagation) [26]. The *testing phase* occurs when the DNN model $F()$ is deployed with fixed parameters θ_n to make predictions on input unseen during training. The weighted parameters represent the DNN knowledge acquired in the *training phase*. What usually happens is that the DNN would generalize and make accurate prediction for new input not seen during training [26].

There are several types of Deep Neural Network architectures, which vary functionality, and dexterity depending on the problems they are designed to solve. Some of the most popular types include (but are not limited to): 1) *Convolution Neural Networks* (CNN), 2) *Recurrent Neural Networks* (RNN), 3) *Sequence-to-Sequence Model* (another type of RNN) which models sequential information in the Time Series sequence , 4)

Auto-Encoders, 5) *Re-enforcement Learning*, 6) *Generative Artificial Networks* (GAN). We do not spend going into depth explaining each type, however, our thesis relies heavily on using a CNN model (as a decoy), but we will explore this architecture further in Chapter 4.

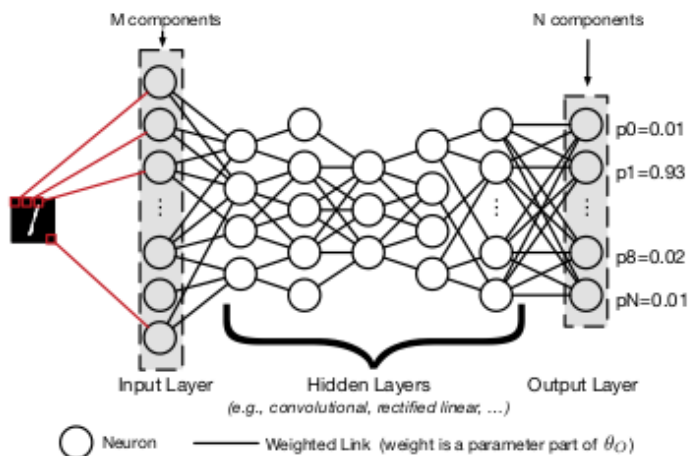


Figure 2.1: A classic DNN architecture. In this particular model, the DNN recognizes images of handwritten digit integers 1...9 and calculates the probability of the image being in one of the $N=10$ classes.

2.1.2 Adversarial Deep Learning

Throughout Deep Learning literature, there have been many works that have focused on deploying deep neural networks, within threatening and malicious environments, potentially exposed to numerous attacks and vulnerabilities [12] [18] [38]. At the center of these threats that attempt to exploit and weaken deep neural nets are *Adversarial Examples*. Please note, because we dedicate an entire section to dissect what these examples are later in the chapter we will focus only on the essential idea behind them. Adversarial Examples are *perturbed* or modified versions of input samples \vec{x} that are used by adversaries to mislead and exploit deep neural networks, during test time, after training of model $F(\cdot)$ is completed [28]. They are crafted with carefully articulated perturbation δX that *forces* the DNN to display a different behavior than intended, chosen by the adversary [28]. It is important to note that the magnitude of perturbation must be kept *small enough* to have a significant effect on the DNN, yet remain unnoticed by a regular human being. These adversarial exploitations vary in their motivations for corrupting a DNN classifier, however some of the most common incentives range from simply reducing

the probability confidence on a target label to a source-label misclassification [28]. Confidence reduction entails reducing the confidence in probability for in accuracy on a label y for a particular input \vec{x} . Contrastingly, Source label classification involves having the model classify an input \vec{x} as a chosen target label, different the original (and intended) source label.

But for any attack to be successful, it requires the adversary to have previous knowledge of the DNN architecture, preferable strong knowledge. This knowledge can perfect (*white-box attack*), partial (*black-box attack*) and no-knowledge (*blind*). However, it is possible to attack a DNN model F with limited knowledge in hand. In works, such as [28], the attacker was able to approximate the architecture of a model F in a black-box setting, and create a substitute model, which was then used to craft adversarial examples on the very same model. These example were transferred back to the model F by cause of *adversarial transferability* [28] - a very powerful property, which enables the attacker to transfer these malicious examples between models evade the classifier.

2.1.3 Deep Learning Threats

While Deep Learning Networks have gathered much attention in terms of capability to solve complicated and hard to solve problems, there are perilous threats that can erode and inhibit their potential [37]. It is believed that deep neural networks can be exploited from these three directions, our thesis focuses on combating the last type:

- *Modified Training Data* - commonly known as a *causative* or *poisoning* attack, in which the adversary influences or manipulates the training data, with a transformation. This modification could entail control over a small portion or an important determinant feature in the data. With this type of attack advance, the attacker can mislead the learner in order to produce a bad classifier, which the adversary exploits post training [36].
- *Poorly Trained DNN Models* - although considered an oversight, rather than blamed on an external adversary, a perfunctory DNN could be due to several reasons. Most of the time, developers user DNNs prepared and trained by others, these same DNNs could become easy targets for manipulation by adversaries [37].
- *Perturbed Input Image* - commonly known as *adversarial examples* [18], attackers are also known to attack DNN models, post testing, by constructing malformed input to evade the learning mechanism learned by the DNN classifier. The latter is known as an *evasion attack* [36].

2.2 Adversarial Examples

In this section, we provide a thorough definition of adversarial examples. Also, relevant in this context, we describe the properties that give adversarial examples their potency, which has earned them their place as one of the most notorious threats to deep learning classifiers. However, we can't discuss adversarial examples without delving into the techniques used to generate them, as well as the impact they have on deep neural networks. We also explore the known defenses techniques against adversarial examples.

2.2.1 Adversarial Examples Definition

As mentioned, machine learning models are vulnerable to adversarial attacks that seek to destabilize the neural network ability to generalize and jeopardize its security and integrity. From what we learned in [36], these attacks can either occur during the training phases as a *poisoning attack* or testing phase as an *evasive attack* on the classification model. In a test-time attack scenario, the attacker actively attempts to circumvent and *evade* the learning process achieved by training the model. This is done by inserting inputs that exploit *blind spots* in the model, which cannot be detected. These disruptive anomalies are known as *Adversarial Examples*. See Figure 2.2 for an illustration of adversarial examples

Adversarial examples are (slightly) perturbed versions of regular input samples to classi-

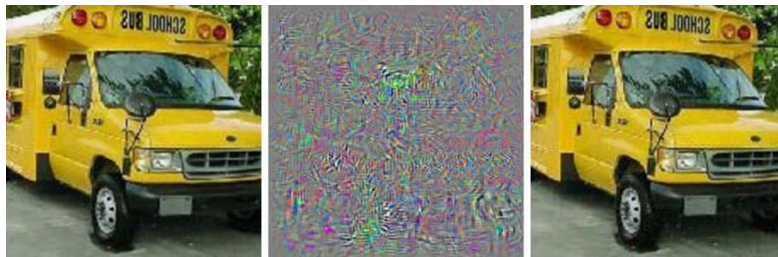


Figure 2.2: Adversarial Examples - Input \vec{x} (left), perturbation ε (middle), adversarial example \vec{x}^* (right)

fiers. They are maliciously designed to have the same appearance as regular input, from a human's point of view, at least. These masqueraded inputs are designed to confuse, mislead, and force the classifier to output the wrong label [15], violating the integrity of the model. These examples can be best thought of as "glitches" that can fool the deep learning model. These glitches are difficult to detect and exploitable, if left unattended. A well crafted adversarial input sample x is inserted into the classifier $A(\cdot)$, the input x was first originally correctly classified. Then, given the same classification model $A(\cdot)$

with input x , so that $C(x) = \ell$, we say x' is an adversarial example if $C(x') \neq \ell$. For an illustration of what adversarial examples are, see the figure 2.3 below for a better understanding of what is constructed by an adversarial example [25]. Classification models are considered *robust* if they are not affected by adversarial examples, which what current adversarial defenses strive for.

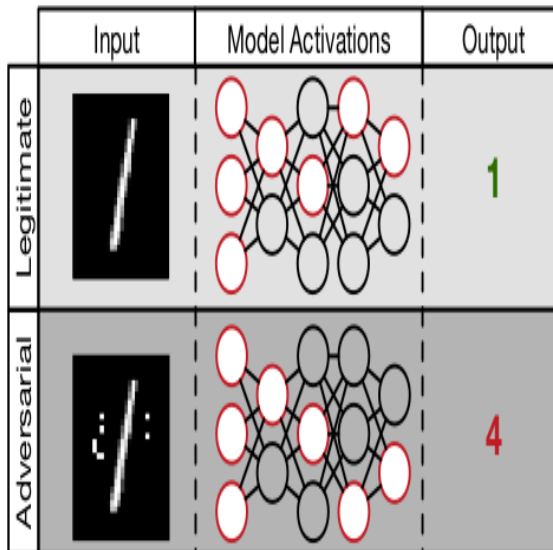


Figure 2.3: An Adversarial Example is generated by modifying a legitimate input sample, in such a way which would cause to the classifier to mislabel it, where as a human wouldn't notice a difference.

2.2.2 Adversarial Example Properties

In the beginning of the previous section (section 2.2.1), we mentioned that adversarial examples x' possess an appearance similar or *close* to the original input samples x , the distance between the original and modified input is known as the p -norm distance. This degree of closeness could be L_2 , which is the *Euclidean Distance* between two pixels in an image, L_∞ , which is the absolute change made to a pixel, or L_1 , which is the total number of pixel changes made to the image [8]. If the measure of distortion in any of the previous metrics of closeness is small, then those images must be visually similar to each other, which makes them a prime candidate for adversarial example generation. In our thesis work, we are assuming the attacker is using L_∞ to measure the absolute change made to a pixel. Another interesting property, which our thesis topic is heavily based on, and which we are trying use with our defense approach is the *transferability*

property. This property states that given two models $F(\cdot)$ and $G(\cdot)$, an adversarial example on F will transfer to G , even if they're trained with two different neural net architecture, or training sets [8]. This intriguing property of transferability enables the attacked model to generalize on the adversarial example, and allow both models $F(\cdot)$ and $G(\cdot)$ to assign the adversarial example to the same class. In order to understand why this property of transferability is possible, we must first understand why it occurs and where these adversarial examples originate from.

Work in [12] shows that adversarial examples are not scattered out randomly but occur in large and continuous regions in the decision space, see Figure 2.4 below for an illustration. The latter property might explain why transferability is possible in this regard. The authors in [12] argue that the higher dimensionality space of adjacent two models, the more likely that the subspaces in these two models will intersect significantly, and an adversarial example will be found that can be assigned to the same class label in both models. Furthermore, the authors in [35] argue that due to the shared dimensionality property, the decision boundary of any two models that have an adversarial example transfer from model $F(\cdot)$ and $G(\cdot)$ must very close to each other for adversarial transferability to be successful.

2.2.3 Adversarial Example Origins

Researchers cannot agree on where these perturbations originate from, or on the more pressing question on simply why they even exist. Many papers have offered hypotheses on their origin. The most popular propositions include work in [12], which state that some models are too *linear* in dimensional data, this could explain why adversarial examples are common in small regions within the decision space. Opposed to the latter, work in [25], explains that adversarial examples could be the result the deep learning model not being "flexible" for certain tasks and inputs. One proponent research *Tramér* suggested that perhaps adversarial examples exist simply due the p -norm distance to the model's decision boundary is in some cases larger than the distance between two models decision boundaries in the same direction.

2.2.4 Generating Adversarial Examples

There are several techniques and methods to generating adversarial examples, used in experiments, such as those in [18] [26]. We refer to the product of these methods as adversarial examples. In order for us to understand how these minatory objects are generated, we must first understand the individual components needed in their creation.

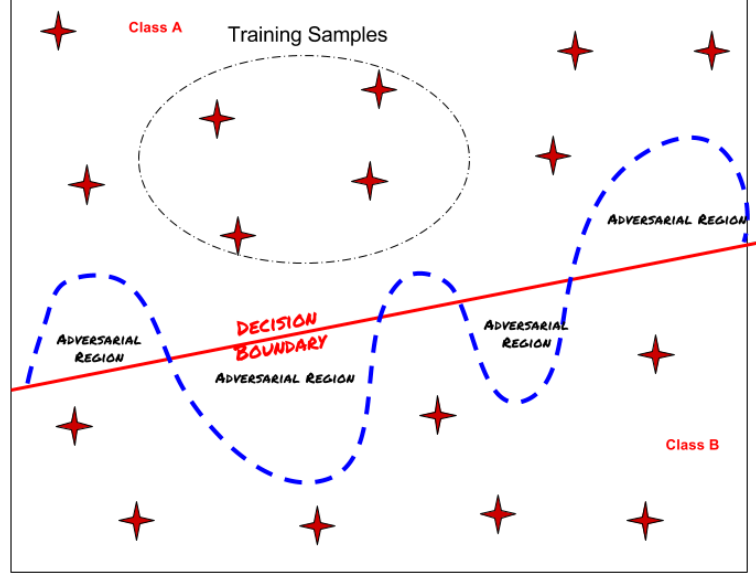


Figure 2.4: Adversarial Regions in Classifier Decision Space

Consider the following notation [18]:

\mathbf{X} - a clean input (untampered) for some dataset D , typically a 3-D tensor (width \times height \times depth). Generally, the image pixel values range between 0 and 255.

$y(\text{true})$ - the corresponding true label for input \mathbf{X} .

$J(\mathbf{X}, \mathbf{y})$ - the cross-entropy cost function used to train the model, given as an input image X , which we wish to maximize the loss function $J()$ for.

ϵ, \mathbf{n} - the hyper-parameters added to influence the model $f(\cdot)$ to create the adversarial example x^* . ϵ represents the magnitude of perturbation in the image, with respect to the metric norm of closeness (L_1, L_2 , or L_∞). We want to keep ϵ *small* to a degree, to remain undetected by the observer.

As mentioned, there are several methods to generate adversarial examples. We won't delve into depth in each method, however we will focus most of our efforts exploring the *Jacobian-based Saliency Map Approach* (JSMA), which the adversary in our thesis uses in his attack. The following are some methods to generate adversarial examples:

Fast Gradient Sign Method (FGSM) - introduced by Goodfellow in [12], and considered to be one of the fastest (and most successful) ways to generate adversarial examples. The idea behind this attack approach is to linearize the cost function $J(\cdot)$ used to train the prediction model by taking the gradient of $f(\cdot)$, with respect to every feature found in the neighborhood of training input samples \vec{x} which the adversary wants to force a misclassification of. The best adversarial example or perturbed image \vec{x}^* is computed from the input \vec{x} by solving the following optimization problem:

$$\vec{x}^* = \vec{x} + \epsilon \text{sign}(\nabla_{\vec{x}} J(x, y(\text{true})))$$

Here, the ϵ represents the factor responsible for calibrating magnitude of perturbation in the input sample \vec{x} , while $J(X, y(\text{true}))$ represents the cost function we wish to maximize for input sample \vec{x} , while keeping ϵ small. The value of ϵ needs to be set carefully since a large value of ϵ to compute the adversarial example will cause the sample to be misclassified by $f(\cdot)$, however will be easily detected by a human, foiling the attack. This method is considered fast because it does not require an iterative procedure to compute \vec{x}^* .

Basic Iterative Method (BIM) - considered to be an extension of the FGSM method above. It is an iterative procedure in the sense that it is applied multiple times with small step size $N+1$ in each iteration, while applying small feature modification to the input sample \vec{x} during each intermediate step. The best adversarial example is computed in its final form by solving the following optimization problem [24]:

$$\vec{x}_{N+1}^* = \vec{x}_N + \epsilon \text{sign}(\nabla_{\vec{x}} J(x, y(\text{true})))$$

Jacobian-based Saliency Map Approach (JSMA) - introduced first by Papernot in [27]. This non-iterative (slower) and computationally intensive generation approach works by perturbing the feature of an input sample \vec{x} that have large *adversarial saliency scores*, used in a targeted attack. Basically, this score represents the goal of taking an sample with important or noticeable features from its source class $f(\vec{x}) = y(\text{true})$ across

the decision boundary to a target class $f(\vec{x}) = t(true)$.

First, the adversary computes the *Jacobian Matrix*, which are all the first-order derivatives and evaluates the inputs. The latter returns a matrix of first-order derivatives $[\frac{\partial f_i}{\partial x_i} \vec{x}]_{i,j}$, where the component i, j is the derivative of class j with respect to input feature i . To computer the saliency map, the adversary computes the following for each matrix item i , given by [24] . See Figure 2.5 below.

$$S(\vec{x}, t)[i] = \begin{cases} 0 & \text{if } \frac{\partial f_t(\vec{x})}{\partial \vec{x}_i} < 0 \text{ or } \sum_{j \neq t} \frac{\partial f_j(\vec{x})}{\partial \vec{x}_i} > 0 \\ \left(\frac{\partial f_t(\vec{x})}{\partial \vec{x}_i} \right) \left| \sum_{j \neq t} \frac{\partial f_j(\vec{x})}{\partial \vec{x}_i} \right| & \text{otherwise} \end{cases}$$

Figure 2.5: Jacobian Saliency Map Approach

The value t is the target class that wish to assign the input sample \vec{x} instead of the source class label $y(true)$. The adversary then selects, from the pool of adversarial samples, a sample i with the highest saliency score $S(\vec{x}, t)[i]$ and maximizes its value. The latter process is repeated until we cause the misclassification to occur on the target class or we reach the maximum number of perturbed features.

There also other adversarial examples generation methods, such as the *iterative-less-likely-class-method* [18]. Other lesser known methods exist such as, *DeepFool*, *CPPN EA Fool*, *C&W's attack* and *BFGS-L attack* mentioned in [38]. However, we will not spend time explaining these methods as they fall out of scope of our thesis.

2.2.5 The Adversarial Optimization Problem

Whether the adversary is generating the adversarial examples using JSMA, or any other of the adversarial example generation methods, one thing is for certain, there is a cost involved. In the general case, adversarial examples are generated by solving a hard optimization problem similar to the one below. Where $\vec{x} + \delta \vec{x}$ represents the least possible

$$\vec{x}^* = \vec{x} + \arg \min \{ \vec{z} : \tilde{O}(\vec{x} + \vec{z}) \neq \tilde{O}(\vec{x}) \} = \vec{x} + \delta \vec{x}$$

Figure 2.6: Adversarial Example Optimization Problem

amount of *noise* added, while remaining unnoticeable by humans. The adversary wishes to produce adversarial examples \vec{x}^* for a specific input sample \vec{x} that will cause a misclassification by the target model T such that, $O\{\vec{x} + \delta \vec{x}\} = O\{\vec{x}\}$. This misclassification

proves that the classifier has been compromised, and is no longer usable. The misclassification error the attacker is after is achieved by adding the least amount of possible noise $\delta\vec{x}$ to the input x , in order to be unnoticeable by humans but just enough to mislead the DNN. This is an optimization problem that is not easy to solve, since it is *non-linear* and *non-convex*. An optimization problem is considered to be *convex* if convex optimization methods can be used on the cost function $J(\theta)$, that if minimized $\min_x J_0(x)$ for the best possible and unique outcome can guarantee a global optimal solution. Here, optimization is likely a well-defined problem here with one optimal solution or *global optimum* across all feasible search regions. On the other hand, a non-convex problem is one where multiple local minimums exist (solutions) exist for the cost function $J(\theta)$. Computationally, it is difficult to find one solution that satisfied all. Here, optimality has become a problem, and an exponential amount of time and variables are needed to find a feasible solution, where many exist. Figure 2.7 illustrates convex vs. non-convex optimization. By preventing the attacker from learning anything about the model T

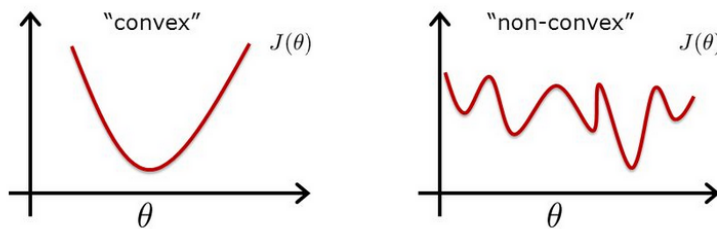


Figure 2.7: Convex vs. non-convex optimization

in a Black-Box System setting; it makes it more difficult to solve this computational challenge. In our approach, we introduce this "difficulty", by deceiving the adversary and allowing him to attempt in solving this optimization problem as an infeasible task for a decoy model, which has no real value. Generating these adversarial examples is exhaustive in computational cost time, as well as approximating and training the substitute decoy model to craft the examples. And if the attacker does indeed succeed in generating these examples, it would be a highly infeasible task done in vain.

2.2.6 Impact of Adversarial Examples on Deep Neural Networks

As it is known, a machine learning application could be in severe jeopardy if the underlying model were to fall in the hands of an adversary, with intentions on launching an attack. However, there are certain measures taken to prevent the latter from occurring,

which are out of scope in this thesis. However, equally menacing, and as likely, is if an adversary were able insert an input, image or query that would bypass the model's learning mechanism, and cause a misclassification attack, in full view of the defender. Adversarial Examples have the ability to do just that. As mentioned in section 2.1, deep neural nets depend on the discriminative features $X_{m,n} = (x_{1,1}, x_{1,2}, x_{1,3}, \dots, x_{1,n})$ embedded within the image that the DNN model recognizes and learns, which it then assigns to its correct class. However, according to [23] it was shown the DNN models can be *tricked* and convinced that a *slightly* perturbed image or input that should be unrecognizable and consequently rejected by the neural network, can be *made* to be generalized and accepted as a recognizable member of a class. The consequence of this is that many state-of-the-art machine learning systems deployed in a real-world setting are left vulnerable to adversarial attacks, at any point in time. This creates calamity, because an input unrecognizable to the model can be classified with high confidence (*false positive*), and an input recognizable to the model can be classified with low confidence (*false negative*), violating the integrity of the model, and making it virtually unusable. There many theories as to why adversarial examples even exist [12] [25], a few things are certain, adversarial examples are common and threaten the stability of deep neural networks.

Some of the most striking examples are in the case of audio inputs that sound unintelligible (to human), but contain voice-command instructions that could mislead the deep neural network [18]. In the case of facial recognition scenario, where the input is subtly modified with markings that a human being would recognize their identity correctly, but the model identifies them as someone else [18].

2.2.7 Combating Adversarial Examples - Defenses

There are numerous countermeasures to defend and fortify a model against adversarial examples, some of which are *reactive*, while others are *proactive*. Some of these methods include augmentation of the training data with adversarial examples, better known as *adversarial training*. The latter essentially works as follows: the model F is trained on both clean as well as perturbed samples. The purpose of this method is to increase the robustness of the training model in all directions, which means it should be able to classify an input sample to its correct class, as well detect any perturbations it may encounter. Below, we briefly explain some of other proposed methods:

- **Network Distillation:** also know as *defensive distillation* [28] [38] originally designed as defense to reduce the size of a DNN by transferring knowing from a

large DNN to a smaller one, to improve robustness. The authors in [28] found that adversary usually target the model’s sensitivity to perturbations, which is what their solution attempted to rectify.

- **Adversarial Re-training:** proposed by Papernot, considered to be another way to make deep neural networks more robust, by regularizing the neural network [12], as well as improving the precision.
- **Input Reconstruction:** adversarial examples can be transformed to clean data, and make them harmless to the deep neural network. The authors in [38] using a *de-noiser* that detects adversarial examples and removes perturbation from the input and converts it to its original form.
- **Classifier Robustifying:** a robust network architecture can help fortify the model and protect it against adversarial attacks. Due to the uncertain nature of adversarial examples, the authors in [38] mention a lot of the models are equipped RBF kernels to improve the distribution of data and robustify the mode against perturbations.
- **Network Verification:** verifying the properties of deep neural network offers a promising path towards robustifying neural networks [38]. This method might give insight into unknown and unseen attacks, that could be prevented pre-training. The authors in [38] suggested using *DeepSafe*, which uses *RELU-plex* in their defense. The authors also suggest using targeted robustness to make the *target class* robust against adversarial attacks.
- **Ensemble Defenses:** this defense approach suggests using multiple defenses, grouped together to curb adversarial attacks. Some examples of ensemble defense include *Pixel Defend* and *MagNet*. However, as mentioned in [38] showed using ensemble defenses does not robustify or make the neural network any stronger.

2.3 Transferability and Black-Box Learning Systems

The section focuses on the concept of Black-Box Learning Systems. We will offer a detailed definition of what a *Black-Box* Threat Model is, as well as how it contrasts from the *Blind* Threat model. We explore the functionality of the Black-box Attack approach, as well the hypothesis responsible for allowing Black-box attacks to occur - *Adversarial Transferability*. This section also offers insight into how Adversarial Transferability is utilized to exploit and launch Black-Box attacks on classification models. Let us first introduce the concept of Adversarial Transferability.

2.3.1 Adversarial Transferability

According to Papernot and Goodfellow, who are known as the proponents of *Adversarial Transferability* in [35], the hypothesis of Adversarial Transferability is formulated as the following:

"If two models achieve low error for some task while also exhibiting low robustness to adversarial examples, adversarial examples crafted on one model transfer to the other."

In simple terms, the idea behind *Adversarial Transferability* is that for an input sample x , the adversarial examples \vec{x}^* generated to confuse and mislead one model m can be *transferred* and used to confuse other models n_{N+1} , that are of homogeneous or even heterogeneous classifier architectures. This mysterious phenomena is mainly due to the property commonly shared by most, if not all machine learning classifiers, which states that predictions made by these models vary smoothly around the input samples making them prime candidates for adversarial examples [15]. Also, it is also worth noting these perturbed samples, referred to here as *adversarial examples*, do not exist in the decision space as a mere coincidence. But according to one hypothesis in [12], they occur within large regions of the classification model decision space. Here, dimensionality of the data is a crucial factor associated with the transferability of adversarial examples. The authors hypothesize that the higher dimensionality of the training data, the more likely that the subspaces will intersect significantly, guaranteeing the transfer of samples between the two subspaces [12].

According to the above hypothesis, transferability holds true between two models *as long as both models share a similar purpose or task* [25]. Knowing this, an attacker can leverage the property of transferability to launch an preemptive attack, by training a local substitute classifier model F on training data that the chosen target classifier T

The figure is a heatmap representing a cross-technique Transferability matrix. The y-axis is labeled 'Source Machine Learning Technique' and the x-axis is labeled 'Target Machine Learning Technique'. Both axes list six techniques: DNN, LR, SVM, DT, KNN, and Ens. The cells in the matrix are shaded in grayscale, with darker shades indicating higher success rates. The numerical values for each cell are as follows:

	DNN	LR	SVM	DT	KNN	Ens.
DNN	38.27	23.02	64.32	79.31	8.36	20.72
LR	6.31	91.64	91.43	87.42	11.29	44.14
SVM	2.51	36.56	100.0	80.03	5.19	15.67
DT	0.82	12.22	8.85	89.29	3.31	5.11
KNN	11.75	42.89	82.16	82.95	41.65	31.92

Figure 2.8: cross-technique Transferability matrix: cell (i,j) is the percentage of adversarial samples crafted to mislead a classifier learned using machine learning technique i that are misclassified by a classifier trained with technique j.

was trained on, with a dataset x of similar dimensions and content, producing adversarial examples \vec{x}^* . It is also worth noting that the success rate of transferability varies depending on the type of target classifier the examples \vec{x}^* are being transferred to. The figure below 2.8 [15] illustrates the transferability matrix for adversarial examples generated with classifier of prediction technique i and transferred to a target classifier trained with technique j . As it can be seen in Figure 2.8 the success rate of transferability is higher in some classification models such as *Support-Vector-Machines* (SVM) and *Linear Regression* (LR), but not others. The latter might be associated with the purity of data samples being used in generating the examples transferred [15].

These modified examples can then be transferred to the target classifier. Hence, the same perturbations that influence model A also effect model B . Knowing that the above hypothesis is true in the general case, Papernot used this very same concept to attack learning systems using adversarial examples generated and transferred from a substitute classifier in [26]. This transfer property is an anomaly, and creates an obstacle in the face of deploying and securing machine learning services on the cloud, enabling exploitations and ultimately attacks on Black-Box systems [35], as we'll see in the coming sections.

To measure the transferability of adversarial examples between m and other models n_{N+1} , we use two points of measurement, which are: 1) *transferability rate*, 2) *success rate*. These two relationships of semblance are used to benchmark the transferability of

the adversarial samples transferred from the substitute model back to the original model [26]. The success rate refers to the portion of perturbations that will be misclassified by the substitute model F , while the transferability refers to those same perturbation samples that will misclassified by the target model, when transferred from the substitute model.

2.3.2 Black-Box Threat Model

To understand what a *Black-Box Threat Model* is, we must first understand what the term *Black-Box* means. A Black-Box is essentially a system that can be construed in terms of inputs x and outputs y , with the internal mechanisms of the system $f(x) = y$ transforming x into y remains invisible. The functionality of the Black-Box can only be understood by observation, which is what the attacker depends on. Figure below 2.9 illustrates a basic Black-Box system.

The Black-Box Threat Model is by extension a Black-Box system. In our thesis, we

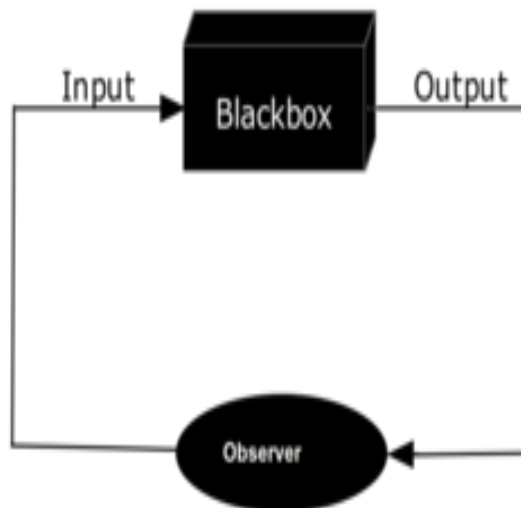


Figure 2.9: A Simple Black-Box System

are attempting to prevent the attacker from polluting the target classifier T , by blocking transferability and access to the target model to change the prediction on the class label. Here, we consider the adversary to be *weak* with limited knowledge, as in he can only observe the inputs inserted and outputs produced, while possessing little knowledge of the classifier itself. The adversary possesses very little, if no knowledge at all of the classifier architecture, structure, number or type of hyper-parameters, activation function, node

weights, etc. Such an environment is considered to be a *black-box system* and the type of attacks are called *black-box attacks*. The adversary need not know the internal details of the system to exploit and compromise it [26].

Generally, in order to attack the model, in Black-Box Learning setting, the adversary attempts to generate adversarial examples, which are then transferred from the substitute classifier F to the target classifier T , in an effort to successfully distort the classification of the output labels [15]. The intension of the attacker is to train a substitute classifier in a way that is to mimic or simulate the decision space of the target classifier. For the latter purpose, the attacker continuously updates the classifier and query the target classifier (Oracle) for labels to train the substitute model, craft adversarial examples and attack the black-box target classifier.

Generally, the model being targeted is a multi-class classifier system, otherwise known as the *Oracle* O . Querying the Oracle represents the only capability which the attacker possesses, which is to query the Oracle for the label $O\vec{x}$ for any input \vec{x} querying the Oracle O . This is the only capability available to the attacker, as in the Black-Box model no access to the Oracle internal details is possible [26].

The goal of the adversary is to produce a *perturbed* version of any input \vec{x} , known as an *adversarial sample* after modification, denoted \vec{x}^* . This represents an attack on the integrity of the classification model (oracle) [26]. What the adversary attempts to do is solve the following optimization problem to generate the adversarial samples, as seen in Figure 2.15 below:

$$\vec{x}^* = \vec{x} + \arg \min \{ \vec{z} : \tilde{O}(\vec{x} + \vec{z}) \neq \tilde{O}(\vec{x}) \} = \vec{x} + \delta_{\vec{x}}$$

Figure 2.10: Adversarial Example Optimization Problem

The adversary must be able to solve this optimization problem by adding a perturbation at an appropriate rate with $\delta\vec{x}$, to avoid human detection. The rate must be chosen in such a way with the least perturbation possible to influence the classifier, as well remain undetected by a human [26]. This is considered a hard optimization problem, since finding a minimal value to $\delta\vec{x}$ is no trivial task, as mentioned in the above section. Furthermore, removing knowledge of the architecture and training data makes it difficult to find a perturbation, where $O\{\vec{x} + \delta\vec{x}\} \neq O\{\vec{x}\}$ [26].

2.3.3 Black-box Threat Model Vs. Blind Threat Model

Although our research mainly focuses on curbing adversarial attacks on black-box learning systems, it is also worth mentioning that other threat models exist. The different

ones differ depending on how the adversary plans to query the substitute model F and attack the target system. We have already been introduced to the first system, as mentioned above, the *Black-Box Threat Model*. While the more constrained second one is the *Blind Model*, which is out of scope.

Opposite to the Black-Box Model, the Blind Model possesses a very limited (small) set of exposed knowledge to the attacker. This limited access applies to the labeled training data and its distribution. Unlike the Black-Box Model, the adversary is blind to the target system he is attacking [15], this means virtually no access. The internal details of the classifier are essentially shielded from the attacker. However, both threat models share some commonalities between them, for instance, Both models involve the attacker training a substitute classifier and use the perturbations generated to attack the target model [15]. They share their innate vulnerability to adversarial attacks, due to Adversarial Transferability, whose effect is more potent in the Black-Box Model than its adjacent Blind Model [15]. Also, the threat model being blind does not prevent it from being exposed to external attacks. Another interesting shared trait is the lack of robustness that the adversarial defense known as *distillation* [28] has inside both models. It was shown that in both models the attacker can evade the effect of its defense method of distillation by adversarial feature blocking which the defense depends on to thwart attacks [7].

2.3.4 Transferability in Black-Box Attacks

Adversarial Transferability is critical for Black-Box Attacks, to say the least. In fact Black-Box systems are dependent on its success. In [37], it is suggested that the adversary can build a substitute model F model with synthetic labels collected by observing the *Oracle* O , despite the DNN model and dataset being inaccessible. The attacker can build a substitute model F from what he learns from O . The attacker will now craft adversarial samples that will be misclassified by the substitute model F [28]. Now that the attacker has approximated the knowledge of the internal architecture of F , he can use it to construct adversarial examples using one of the method described in section 2.3.3. For as long as adversarial transferability holds between F and O . adversarial examples misclassified by F will be misclassified by O . In our thesis, we must find a way to block adversarial transferability from occurring, and we plan to accomplish this via deception.

2.3.5 Transferability of Adversarial Examples in Black-Box Attacks

It was Papernot in [26] [25] who proposed that transferability can be used to transfer adversarial examples from one neural network to the other that share a common purpose or task, yet are dissimilar in network architecture. Transferability is essential for the success of Black-box attacks on deep neural nets, which is due to the limitations imposed on the adversary, such as lack of architecture, model and training dataset. Even with limited knowledge, the adversary with the aid of the transferability property in the adversary's armaments, the adversary can train a substitute model and generate transfer the examples against it, then transfer them to the target model, making the victim's trained model vulnerable to attack [38].

There has been much work focused on the abilities possessed by adversarial examples, and its ability to transplant itself between machine learning techniques (DNN, CNN, SVM etc) bypassing situated defense methods, namely work in [8] [20] [26], which all reached the same conclusion - adversarial examples will transfer across different models trained on different dataset implemented, with different machine learning techniques.

2.3.6 Black-Box Attack Approach

As mentioned in the section 2.5.2, the adversary wishes to compromise the integrity of the classification model by querying the labels provided by the *Oracle* O for the input \vec{x} . The adversary's plan is use the labels, collected by observing the *Oracle* O to generate a substitute model F . The adversarial goal of finding a minimal perturbation to misclassify a targeted classifier model is a difficult problem (section 2.2.5), which happens to be non-convex, where multiple solutions exist for the global minimum of the optimization problem [26]. What the adversary can do is attempt mimic or approximate the *Oracle* O model architecture, but this requires internal knowledge of the classifier internal structure, which is not possible considering that inaccessibility under a Black-Box model scenario [26]. Another benefit to this approach is the fact that most machine learning models require large and expensive training datasets. This makes incredibly difficult for the attacker to attack a system in such a environment [26]. The Black-Box attack strategy consists of the following steps:

Substitute Model Training:

Here, the attacker queries the *Oracle* O with synthetic inputs generated using one of the adversarial samples selected by one of the adversarial training algorithms to build a substitute model F , which will be used to misclassifying the target model due to the

adversarial transferability property [26]. The notion of creating a substitute model F is considered challenging due to two main reasons: 1) selecting an architecture F is difficult since we have no knowledge of the *Oracle* O model, 2) the number of queries to the *Oracle* O is limited to remain undetected [26]. The authors in [26] emphasize that their Black-Box approach is not meant to increase substitute model F accuracy, but to approximate the decision boundaries as best as possible, with few labels [26].

Substitute Architecture:

According to the authors in [26], this step is considered the most limiting factor when it comes to attacks on Black-Box Learning systems. This is due to the notion that without knowledge of the target model architecture, the attacker knows very little about how the system processes input (text, images, or media) and produces output (label or probability vector). One potential way, suggested by the authors, to select an appropriate architecture for the substitute model is to simply explore and try different variations of the substitute architectures and select one that yields that highest level of success [26].

Generating Synthetic Architecture:

Another complication in the path of building a successful Black-Box attack is the dataset used to train the substitute model F . We could potentially request an infinite number of queries to get the oracle's output $O\vec{x}$ for an input sample \vec{x} [26]. However, this method, although effective in the sense to create a copy of the Oracle is actually not methodical, this is due to the excess *Oracle* O querying. Creating a large number of queries attracts attention from the defender and makes the adversarial attempts to detect, and ultimately deflect [26].

Substitute Model Training:

Here, we describe the five step algorithm procedure described in [26]. The algorithm is outlined in full below in figure 2.11 below. For a better understanding of how this algorithm is used to train substitute model, see below for a brief description of the steps involved [26]:

Initial Collection (step 1): the adversary collects a small set of input samples that representative of the *Oracle* O input domain. These inputs do not need to necessarily come from the same statistical distribution.

Algorithm 1 - Substitute DNN Training: for oracle \tilde{O} , a maximum number max_ρ of substitute training epochs, a substitute architecture F , and an initial training set S_0 .

Input: $\tilde{O}, max_\rho, S_0, \lambda$

- 1: Define architecture F
- 2: **for** $\rho \in 0 .. max_\rho - 1$ **do**
- 3: *// Label the substitute training set*
- 4: $D \leftarrow \{(\vec{x}, \tilde{O}(\vec{x})) : \vec{x} \in S_\rho\}$
- 5: *// Train F on D to evaluate parameters θ_F*
- 6: $\theta_F \leftarrow \text{train}(F, D)$
- 7: *// Perform Jacobian-based dataset augmentation*
- 8: $S_{\rho+1} \leftarrow \{\vec{x} + \lambda \cdot \text{sgn}(J_F[\tilde{O}(\vec{x})]) : \vec{x} \in S_\rho\} \cup S_\rho$
- 9: **end for**
- 10: **return** θ_F

Figure 2.11: Substitute Model Training

Architecture Selection (step 2): the adversary selects an architecture to be trained as the substitute model F .

Substitute Training the adversary selects more appropriate substitute models F_p by repeating the following steps:

Labeling (step 3): the labeled output $\tilde{O}\vec{x}$ is collected by querying the *Oracle* O . These labels are then used to compose the substitute model training set S_p to train the substitute model F .

Training (step 4): the adversary trains the substitute architecture model selected in step 2 above and trained using known adversarial training techniques, using labeled data S_p collected from step 3.

Augmentation (step 5): the authors augmentation technique in [26] is used on the training set S_p in order to produce a much larger training set S_{p+1} with more

training data points. step3 and step4 are repeated for the augmented dataset. Step 3 is repeated several times to increase the substitute model F accuracy and mimic the decision boundaries of the *Oracle* O . Here, λ is the parameter of augmentation, which represents the step taken in the direction to augment from S_p to S_{p+1} .

2.3.7 Defense Strategies Against Black-Box Attacks

According to [26], there are two main methodologies which aim to defend against Adversarial Attacks in Black-box systems. The first one is *reactive* and the second is *proactive*. Here, reactive refers to defenses where the defender seeks to detect adversarial examples, while proactive refers to defenses where the defender seeks to make the classifier more robust. Some of the known defense strategies used to curb adversarial attacks in [15] include: 1) Preprocessing Methods 2) Regularization and Adversarial Training, 3) Distillation Methods, and 4) Classification with Rejection. Let us review a few of these methods:

Preprocessing Methods:

The authors in [28] mention this method as a way to filter out input determined to be adversarial. The authors argue that images have natural properties such as high correlation between adjacent pixels or low energy in high frequency. Assuming that adversarial examples and regular input do not lie in the same decision spaces can be used as a pre-text used to filter out malicious perturbations. According to the authors, this method is possibly not the best way to defend against adversaries since the process of filtering could potentially reduce the classifier accuracy on harmless input samples, not deemed adversarial.

Regularization and Adversarial Training:

The authors have suggested using Regularization, Adversarial Training or smoothing as a technique to robustify the classifier against adversarial attacks. The authors in [12] mention one experiment where the accuracy of the classifier fell to 17.9% with adversarial training. This type of defense cannot be relied upon when deploying security sensitive machine learning services. Regularization and Adversarial training has been shown in [15] to be effective for both Black-Box and Blind Threat Models against adversarial examples.

Distillation Methods:

in [28] Papernot proposed using a method called defensive distillation to counter adversarial examples in a Black-Box setting. This method proved useful since it limits the attacker’s ability to select adversarial examples. However, there has been research which indicates that the effect of distillation can be reverted, such as work in [7]. This was made evident in the previous section where it was shown that in both threat models (Black-Box and Blind), the attacker can evade the effect of the defense method of distillation of adversarial feature blocking which the defense method depends on to thwart attacks [7]. The authors suggest that this lack of robustness is due to that notion that defensive distillation only works in the general case, but fails to protect the classifier in cases where the features are all modified at once.

2.4 Honeypots

The section focuses on the concept of *Honeypots*, we'll start with a basic definition of what a *honeypot* is. Then, we dissect and explain the different types of honeypots and evaluate each types' intrinsic value, as well as the different deployment types. This section also offers insight into how other security researchers have proposed using honeypots as a tool in infrastructure security and protection, as well as how we plan to use it in our thesis.

2.4.1 Concept of Honeypots

A honeypot can be thought of as a single or group of fake systems to collect intelligence on an adversary, by inducing him/her to attack it. A honeypot is meant to appear and respond like a real system, within a production environment. However, the data contained within the honeypot is both falsified and spurious, or better understood as *fake*. A honeypot has no real production value, instead its functionality is meant to record information on malicious activity. In the scenario that it should become compromised it contains no real data and therefore poses no threat on the production environment [19] [34]. As mentioned, honeypots can be deployed with fabricated information, this can be an attractive target to outside attackers, and with the correctly engineered characteristics can be used to re-direct attackers towards decoy systems and away from critical infrastructure [13]. See below for a typical architectural design of a honeypot system in Figure 2.12.

2.4.2 Classification of Honeypots

Honeypots can be classified using several different criteria. However, for purposes of this thesis we classify them based on functionality and operation.

- **Research Honeypots** - they're honeypots deployed with the highest level of risk associated with them, this is in order to expose the full range of attacks initiated by the adversary. They're mainly used to collect statistical data on adversarial activities inside the honeypot [19]. They're more difficult to deploy, but this does not hinder from their use by organizations to study attacks and develop security countermeasures against them. Research Honeypots help understand the trends, strategies and motives behind adversarial attacks [22].
- **Production Honeypots** - they are honeypots known for ease of deployment and utility, and use in company production environment [22]. Closely monitored

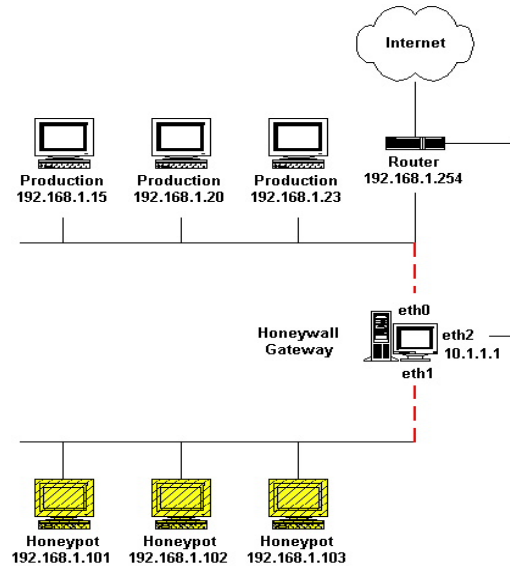


Figure 2.12: Classic Honeypot Architecture

and maintained, their purpose lies in their ability to be used in an organization's security infrastructure to deflect probes and security attacks. They're attractive as an option for ease of deployment and the for sheer value of information collected on the adversary.

- **Physical/Virtual Honeybots** - physical honeypots are locally deployed honeypots, being part of the physical infrastructure. considered to be intricate and difficult to properly implement [19]. On the other hand, virtual honeypots are simulated systems (virtualized) by the host system to forward network traffic to the virtual honeypot [22].
- **Server/Client Honeybots** - the main different between server and client honeypots is the former will wait until the adversary initiates the communication, while client Honeypots contact malicious entities and request an interaction [22]. However, traditional honeypots are usually server-based.
- **Cloud Honeybots** - they are honeypots deployed on the cloud. This type of honeypot has many advantages, as well as restrictions. They are used by companies that at least have one part of their infrastructure on the cloud. Having the system (or part of it) in the cloud has its advantages, it makes it easy to install, update, as well as recover the honeypot in case of a corruption [19].

- **Honey-tokens** - can be thought of as a *digital* pieces of information. It can manifested from a document, database entry, E-mail, or a credentials. In essence, it could be anything considered valuable enough to lure and bait the adversary. The benefit with these tokens is that they can be used to track stolen information and level of adversarial abuse in them system [5]

2.4.3 Honeyplot Deployment Modes

Honeyplots can be deployed in one of three deployment modes [6], they are:

- **Deception** - this mode manipulates the adversary into thinking the responses are coming from the actual system itself. This system is used as a decoy and contains security weaknesses to attract attackers. According to researchers, a honeypot is involved in deception activities if its responses can deceive an attacker into thinking that the response returned is from the real system.
- **Intimidation** - this mode used when the adversary is aware of the measures in place to protect the system. A notification may inform the attacker that the system is protected and all activity is monitored. This countermeasure may ward or scare off any adversarial *novice*, and leave only the experienced adversaries with in-depth knowledge and competent skills to attack the system.
- **Reconnaissance** - this mode is used to record and capture new attacks. This information is used to implement heuristics-based rules that can be applied in intrusion detection and prevention systems. With reconnaissance, the honeypot is used to detect both internal and external adversaries of the system.

2.4.4 Honeyplot Role and Responsibilities

The true value of honeypots lay in their ability to address the issue of security in production system environments, they mainly focus ons:

- **Interaction** - the honeypot should be responsible for interacting with the adversary. this pertains to acting as the main environment where the adversary becomes active and executes his attack strategy.
- **Deception** - the honeypot should be responsible for deceiving the adversary. This pertains to the disguising itself as a normal production environment, when in fact it's a *trap* or *sandbox* designed to exploit the adversary.

- **Data Collection** - the honeypot should be responsible for capturing and collecting data on the adversary. This information will potentially be useful for studying the attacker and his motivations.

Advantages of Honeypots

Honeypots, alone, do not enhance the security of an infrastructure. However, we can think of them as subordinate to measures already in place. However, this level of importance does not take away from some distinct advantages when compared to other security mechanisms. Here, we highlight a few [22]:

- **Valuable Data Collection** - honeypots collect data which is not polluted with noise from production activities and which is usually of high value. This makes data sets smaller and data analysis less complex.
- **Flexibility** - honeypots are a very flexible concept to comprehend, as can be seen by the wide array of honeypot software available in the market. This indicates that a well-adjusted honeypot tool can be modified and used for different tasks, which further reduces architecture redundancy.
- **Independent from Workload** - honeypots do not need to process traffic directed or which originates from them. This means they are independent from the workload which the production system experiences.
- **Zero-Day-Exploit Detection** - honeypots capture any and every activity occurring within them, this could give indication to unseen adversarial strategies, trends and zero-day-exploits that can be identified from the session data collected.
- **Lower False Positives and Negatives** - any activity that occurs inside the server-honeypot is considered to be out-of-place and therefore an anomaly, which is by definition an attack. honeypots verify attacks by detecting system state changes and activities that occur within the honeypot container. This helps to reduce false positives and negatives.

Disadvantages of Honeypots

Ultimately, no one security system or tool that exists is faultless. Honeypots suffers from some disadvantages, some of them are [22]:

- **Limited Field of View** - a honeypot is only useful if an adversary attacks them, and worthless if no one does. If the honeypot is evaded by the adversary, and attacks the production system or target environment directly, it will not be detected.
- **Being Fingerprinted** - here, fingerprinting signifies the ability of the attacker to identify the presence of a honeypot. If the honeypot behaves differently than a real system, the attacker might identify and consequently detect it. If their presence is detected, the attacker can simply ignore the honeypot and attack the targeted system instead.
- **Risk to the Environment** - honeypot might introduce a vulnerability to the production infrastructure environment, if exploited and compromised. And naturally, as the level of interaction (freedom) that the adversary has within the environment increases, so does the level of potential misuse and risk. The honeypot can be monitored, and the risk mitigated, but not completely eliminated.

2.4.5 Honeypots Level of Interaction

A honeypot is considered to be a fake system, with no real value. It is built and designed to emulate the same tasks that a real production system can accomplish. However, these tasks are of no significance, hence compromising the honeypot poses no threat on the production environment. Honeypot systems functionality can be categorized according to the level of interaction the adversary has with the honeypot system environment [19]:

- **Low-interaction Honeypot(LIHP)** - these types of systems emulate only simple services like *Secure Shell* (SSH), *Hypertext Transfer Protocol*(HTTP) or *File Transfer Protocol*(FTP). These systems are easily discoverable by attackers and provide the lowest possible level of security. However, they have a promising advantage, they are easy to install, configure and monitor. They should not be used in production environments, but for education and demonstration purposes. Some examples of such systems include *Honeyperl*, *Honeypoint*, and *mysqlpot*. See a typical architectural design of a low-interaction honeypot in Figure 2.13 below.
- **Medium-interaction Honeypots(MIHP)** - this type of system is a hybrid, which lays in the middle ground between low/high interaction honeypots. This means that the honeypot is still an instance that runs within the operating system. However, it blends in so seamlessly into the environment that it becomes difficult to detect by attackers lurking within the network. Some examples of such systems are *Kippo* and *Honey.py*.

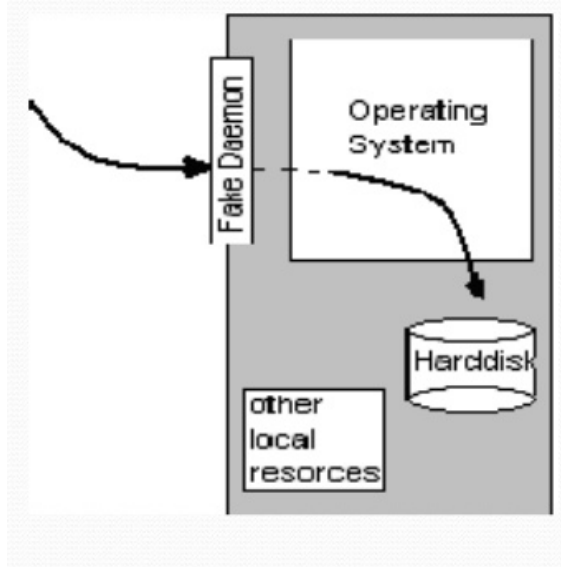


Figure 2.13: Low Interaction Honeypot

- **High interaction Honeypot(HIHP)** - the main characteristic regarding High-Interaction Honeyspots is that they're using a real live operating system. It uses more hardware resources and poses a major level risk on the rest of the production environment and infrastructure, when deployed. In order to minimize risk and prevent exploitation by an adversary, it is constantly under monitoring. Some examples of such systems are *Pwnypot* and *Capture-HPC*. See a typical architectural design of a high-interaction honeypot in Figure 2.14 below.

2.4.6 Uses of Honeyspots

As mentioned above, honeypots have a wide array of enterprise applications and uses. Currently, honeypot technology has been utilized in detecting *Internet of Things* (IoT) cyberattack behavior, by analyzing incoming network traffic traversing through IoT nodes, and gathering attack intelligence [9]. In robotics, a honeypot was built to investigate remote network attacks on robotic systems [16]. Evidently, There is an increasing need to install *red herring* system in place to thwart adversarial attacks before they occur , and cause damage to production systems.

One of the most popular type of honeypots technologies witnessing an increase in its popularity is *High-Interaction-Honeypots* (HIHP). This type of honeypot is preferred since it provides a real-live system for the attacker to be active in. This property is valuable, since it captures the full spectrum of attacks launched by adversaries within

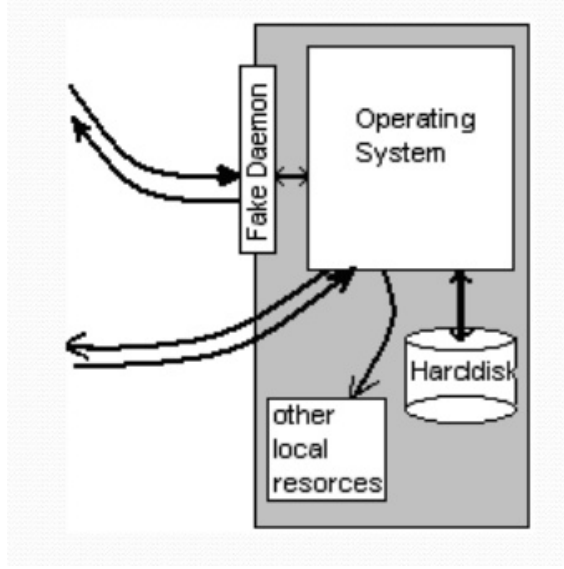


Figure 2.14: High Interaction Honeypot

the system. It us allows to learn as much as possible about the attacker, the strategy involved and tools used. Gaining this knowledge allows security experts to get insight into what future attacks might look like, and better understand the current ones.

In the next chapter, we will explore and discuss the work done by other researchers in the areas of Black-Box systems defense, as well as work in deception as a method of defense.

2.5 Honeypot in our Solution

We formulate the problem of devising an supplementary method of defense against Adversarial Examples. This secondary level of defense will shield the Black-Box System, using honeypots as the primary tool of deception in building the system.

This decentralized and decentralized framework must consist of N of high-interaction honeypots. Each of these N honeypots is embedded with a decoy target model \tilde{T} designed to lure and prevent an adversary with adversarial input $O'\vec{x}$ from succeeding in causing a mislabeling attack on the target model T . Essentially, the framework must perform the following tasks below.

Firstly, prevent the adversary from mimicking the neural network behavior and replicating the decision space of the model. This will be done by essentially blocking adversarial

transferability, prevent the building of the substitute model F from occurring, and the transfer of samples from the substitute model F to the target model T . This makes it difficult to find a perturbation that satisfied $O\{\vec{x} + \delta\vec{x}\} = O\{\vec{x}\}$

Secondly, the framework must lure the adversary away from the target model T , using deception techniques. These methods consist of using: 1) deployment of uniquely generated digital breadcrumbs (HoneyTokens), 2) setting weak TCP/IP ports open to attract the adversary scanning the network, and 3) set up decoy target models \tilde{T} , deployed inside the honeypots for the attacker to interact with, instead of the actual target model T .

Finally, create an infeasible amount of computational work for the attacker, with no useful outcome or benefit. This can be accomplished by presenting the attacker with the non-convex, non-linear, and hard optimization problem, which is generating adversarial samples to transfer to the target model, which in this case is a decoy; a decoy of the same optimization problem below:

$$\vec{x}^* = \vec{x} + \arg \min\{\vec{z} : \tilde{O}(\vec{x} + \vec{z}) \neq \tilde{O}(\vec{x})\} = \vec{x} + \delta_{\vec{x}}$$

Figure 2.15: Adversarial Example Optimization Problem

This strenuous task is complicated further for the attacker because in order to generating the synthetic samples, the attacker must approximate the unknown target model architecture and structure to train the substitute model F , which is challenging. Evasion is further complicated as the number of deployed honeypots N in the framework increases. Therefore, building this system consists of solving three problems in one, preventing of adversarial transferability, deceiving the attacker and creating immense computational work for adversary targeting the system to waste the adversary time and resources. All the later, while keeping the actual target model T out of reach.

Chapter 3

Related Work

The purpose of this chapter is to summarize and evaluate the authored work of other researchers on the techniques and frameworks designed to defend Black-box systems from adversarial example attacks. This chapter also touches on how deception, as a defense technique, is used to protect security systems, specifically with the use of fake digital entities to attract, lure and deceive adversaries.

The works below focus directly on the concept of defending against adversarial examples, aimed at misleading the classifier. moat of the known defense methods are accomplished by data preprocessing and *sanitation* during the training phase of the neural net model preparation. That latter typically means influencing the effect that data will have on the underlying DNN model, and filtering out malicious perturbations inserted by an adversary that may corrupt the classifier, and cause a misclassification. Other notable works in this section focus on the role of cyber security defense through method of deception, specifically on the role of decoys and fake entities to deceive the attacker. Our challenge here, lays in constructing a secondary level of protection and defense, designed not to replace existing techniques, but supplement and reinforce the aforementioned defense frameworks below, through means of adversarial deception.

An alternative (but conceptually close) to the data preprocessing technique is the use of *distillation as a defense* against adversarial perturbations [28], to reduce the impact impurities make on the neural net features. Here, the authors use this simple technique to reduce the dimensionality of DNN classifier model, while maintaining accuracy. However, the work in [7] suggest that a model protected using defense distillation is no more secure than a model without any defense. While the latter is able to defend against some cases of attack, it cannot protect the model's neural net against all types of attack that occur simultaneously, targeting different features .

The following papers and works deal directly with defenses against adversarial examples and other works associated with defense through deception using HoneyTokens:

- i. *Efficient Defenses Against Adversarial Attacks* - this paper [39] focuses on addressing the lack of efficient defenses against adversarial attacks that undermine and then fool Deep Neural Networks (DNNs). The need to tackle this hurdle has been amplified by the fact that there isn't a unified understanding of how or what makes these DNN models so vulnerable to attacks by adversaries. The authors propose an effective solution which focuses on re-reinforcing the existent DNN model and making it robust to adversarial attacks, attempting to fool it. The proposed solution focuses on utilizing two strategies to strengthen the model, which can be used separately or together. The first strategy is using bounded ReLU activation functions in the DNN architecture to stabilize the model, and second is based on augmented Gaussian data for training. Defenses based on data augmentation improves generalization since it considers both the true input and its perturbed version. The latter enables a broader range of search in the input, then say, adversarial training, which is limited and only uses part of the input, falling short. The result of applying both strategies results in a much smoother and stable model, without losing on the model's performance or accuracy.
- ii. *Blocking Transferability of Adversarial Examples in Black-Box Learning Systems* - this paper [15] is the most relevant academic paper, with regards to motivation and stimulus for purposes of developing our proposed auxiliary defense technique with honeypots. The authors in [15] propose an training approach aimed at robustifying Black-Box learning systems against adversarial perturbations, by blocking transferability, which covertly transfers perturbed input between classifiers. The proposed method of training called *NULL labeling*, works by evaluating input and lowering confidence on the output label if suspected to be perturbed and rejecting the invalid input, which spars out of the data distribution of the training data. The training method smoothly labels , filters out and discards the invalid input (NULL), which does not resemble training data, instead of allowing it to be classified into intended target label. The ingenuity of this approach lies in how it is able to distinguish between clean and malicious input. This method proves its capability in blocking adversarial transferability and resisting the invalid input that exploit it. The latter is achieved by mapping malicious input to a NULL label and allowing clean test data to be classified into its original label, all while maintaining prediction accuracy.

- iii. *Towards Robust Deep Neural Networks with BANG* - this paper [31] is another training approach for combating adversarial examples and robustifying the learning model. The authors propose this technique in response to the unknown reason for the existence of adversarial examples, not to mention their abnormal and mysterious nature, which has become a concern for security. The authors argue that regular adversarial training still makes the model vulnerable and exposed to adversarial examples. For this very purpose, the authors present a data training approach, known as *Batch Adjusted Network Gradients* or *BANG*. This method works by attempting to balance the causality that each input element has on the node weight updates. This efficient method achieves enhanced stability in the model by forming *smoother* areas concentrated in the classification region that have classified inputs correctly and become robust against the input perturbations, that work on exploiting and violating the integrity of the model. This method is designed to avoid instability brought about by the adversarial examples pushing the misclassified samples across the decision boundary into the incorrect class. This training method achieves good results on DNNs with two distinct datasets, and has low computational cost, while maintaining classification accuracy on both models.
- iv. *HoneyCirculator: distributing credential HoneyToken for introspection of web-based attack cycle* - in this paper [5] the authors suggest a framework that actively and purposely leaks digital entities in the network to deceive the adversaries and lures them to a honeypot, which is actively under monitoring, tracking token access, and watching out for new adversarial trends. In a period of 1 year, the monitored system was compromised by multiple adversaries, without being identified as a ruse. This framework allows the constant surveillance of adversaries, without being recognized as a decoy. The authors argue that this method is successful, as long as the attacker does not change his attack strategy. However, a main concern for the authors is designing *convincing enough* fake data to deceive, attract, and fool an adversary. The authors argue that the defender should design fake entities that are *attractive* enough to bait the attacker, while not revealing important or compromising information to the attacker, and learn as much as possible about the attacker. As the threat of adversarial attacks increases, so will the need for novelty in the approach to combat it.
- v. *A Survey on Fake Entities as a Method to Detect and Monitor Malicious Activity* - this survey paper [29] serves as an examination of the concept of *fake entities* and digital tokens, which my thesis defense relies partially focuses on. Fake

entities, although primitive, are an attractive asset in any security system. The authors suggest fake entities could be files, interfaces, memory, database entries, meta-data, etc. For the authors, this inexpensive, lightweight, and easy-to deploy *pawns* is as valuable as any of the other security mechanisms in the field such as firewalls or packet analyzer. Simply put, they're digital objects, embedded with fake divulged information, intended to be found and accessed by the attacker. For the authors, operating system based fake entities are the most attractive and fitting to become a decoy, due to the variety of ways the operating system interface can be configured. Once in possession by the attacker, the defender is notified and can begin monitoring the attacker's activity. The authors implemented a framework that actively leaks credentials and leads adversaries to a controlled and monitored honeypot. However, the authors have yet to build a proof of concept.

There is also extensive work done on utilizing adversarial transferability in other forms of adversarial attacks, deep learning vulnerabilities in DNNs, and black-box attacks in machine learning. Among other interesting works that served as a motivation for this thesis include: utilizing honeypots in defense techniques, such as design and implementation of a honey-trap [10], deception in decentralized system environments [33], and using containers in deceptive honeypots [17].

As mentioned, our approach, using honeypots, does not seek to replace any of the existing methods to combat adversarial examples in black-box setting. However, it can effectively be used as an auxiliary method of protection that *laminates* existing defense methods. below, we briefly outline the limitations in our method, as well as how it differs from other defense methods and techniques in the following ways:

Chapter 4

Proposed Defense Approach

In this chapter we introduce an architecture for a first-of-its-kind decentralized defense framework geared towards reducing risk and combating adversarial examples within a context of a Black-Box attack setting. This proposed framework - *Adversarial Honeypots* lays a foundation for a superimposing defense system that blankets existing robust adversarial defense techniques, such adversarial training or distillation. at its core, it utilizes the computer security tool, known as high-interaction honeypots filled with fabricated information to deceive, lure, exploit and eventually learn from the actions of the adversary. This plan of artful misrepresentation and swindle is aimed at blocking adversarial transferability from being used to transfer maliciously perturbed examples perpetrated by the attacker, as well as prevent a consequent misclassification attack from violating the target model's integrity. Although the aforementioned framework increases the defense costs if the attacker suspects a trap and aborts his exploit, there is still a high chance that an attacker will not disregard it. The latter is one of the perilous assumption we have made while designing this system

To the best of our knowledge, this is the first time high-interaction honeypots have been used to address the issue of adversarial attacks in machine learning. As mentioned in the previous chapter the proposed framework has the following advantages: 1) preventing target model interaction by an adversary through means of deception to block adversarial transferability from occurring leading him away from the target model T_{target} towards a decoy model T_{decoy} ; 2) a fail-safe to supplement less-than-secure defense techniques which works by augmenting existing security measures by enticing the adversary using elaborating deception techniques in a sequence. This method does not add any extra complexity, but simply obstructs efforts and fazing adversaries; 3) adversarial information reconnaissance through the use of high-interaction honeypots, the data collected

from which can be used adversarial motives and techniques. Also, this proposed system is highly implementable, but due to time and resource constraints we were only able to implement only one integral component.

This chapter starts with Section 4.1; a formulation of the problem definition, motivations and requirements responsible for the conception of this system. We also highlight the design decisions and assumptions made prior to building this system in the later section. Section 4.2 presents and discusses the techniques behind this defense approach, such as the use of honeytokens and a decoy target model. Section 4.3 provides a technical depiction of the adversary's attack objectives, prior adversarial knowledge and attacker capabilities.

4.1 Problem Definition

what we discussed in related work

What the problem is technically

Why is this problem

4.2 Assumptions

Adversarial Knowledge - We construct a Black-box attack environment by assuming the following about what knowledge the adversary is bounded by, shaping the attack model. The attacker has limited or little 1) surrogate decoy DNN testing pair dataset (x', y') sampled from the same distribution as the training set; 2) queries allowed to ask the *Oracle* O , $q = \{q_1, q_2, q_3, \dots, q_n\}$; 3) data features representation $f = \{f_{1,1}(f_{1,2}, f_{1,3}, \dots, f_{1,n}), f_{2,1}(f_{2,2}, f_{2,3}, \dots, f_{2,n}), f_{3,1}(f_{3,2}, f_{3,3}, \dots, f_{3,n}), \dots, f_n(f_{n,n+1} \dots)\}$. We also assume the attacker has partial knowledge of the following, 1) purpose of DNN model; 2) DNN input and output layers of the DNN, represented by $X = \{X_1, X_2, X_3, \dots, X_n\}$ and $Y = \{Y_1, Y_2, Y_3, \dots, Y_n\}$ respectively; 3) existence of honeypot (disguised as production server) system weak ports for easy access entry.

Honeypot Node Compromise and System Expropriation - It is reasonably sound to assume that no system is temper- resistant. This leads us too believe that one or any of the honeypot nodes in the decentralized framework can become compromised, at any point in time during an adversarial attack. The following are only some of the possible worse-case scenarios: 1) overwhelming the node, embedded with the DNN

decoy model T_{Decoy} with query requests q_{n+1} causing a type of DDoS attack, this can be handled by limiting or setting a threshold on the number of queries an adversary can send to the *Oracle* O , as well as limit the number of queries the *Oracle* O can accept per session. However, certain counter-measures need to be set, in order prevent the case where an adversary sends connects/queries to more than one DNN decoy model T_{Decoy_1} , T_{Decoy_2} to build the substitute training-set; 2) Another possibility is the falsification of communication messages between the different honeypot nodes in the network topology. Although a signed certificate and public/private key (Diffie-Hellman key exchange for example) can handle this issue. However, we should never underestimate the attacker, as he could get access and override security entries in the Sampa database, which is why administrative authorization and certificates should be required to change any entries in the data log.

Deception in Defense As a Proxy - as oppose to the frameworks in [39] [15] [21], which focus on curbing adversarial attacks by *normalizing* the input samples \vec{x} injected into the target model T , our defense framework follows a different method of protection. As mentioned in the novelty of approach (section 3.1), we attempt to deceive the adversary by luring him away from the target classification model T_{Target} , to a decoy model *decoyed* replica T_{Decoy} , deployed within a honeypot node. Inside this environment, the attacker interacts and queries a decoy Oracle O_{Decoy} and build a substitute model F similar to T_{Decoy} , then using the samples from F and transfers them to model T_{Decoy} , and cause a targeted misclassification $T(x) = y(false)$. Our method of defense must be used as a supplemented or proxy-tier of protection. This is imperative since alone it maybe rendered ineffective if the adversary is not deceived or duped by decoy. But deployed in conjunction with a different weak defense method or one which uses reinforcement learning can be a powerful cohort to help in boosting defensive measures against adversarial attacks.

Adversarial Token Plausibility - The adversarial honeytokens $H_n = \{H_1, H_2, ..., H_n\}$ generated by our framework are meant to be designed with high subjectivity in mind. Articulately, they must appear convincing to lure outside adversaries and match their exploitive intentions. These items must be fascinating to the adversary, otherwise there is no inclination to reasonably believe in their apparent authentic contents, or if they were leaked by mistake. Believability is intuitive and would vary on a per adversarial attacker basis, making difficult to simulate. Deception in this case is personalized, this means the same adversarial token might trick one adversary but not the other. These

adversarial tokens were not meant to be generated generically. In consequence, the defender must have a thorough understanding of the adversary's nature and intentions in order to design an individualized digital token, targeted at the attacker. This can be an issue especially if the learning model being attacker is by dynamic adversary, who never attacks or uses the same attack technique twice. It becomes more difficult to artificially simulate legitimate objects as adversaries become more cunning and cautious.

Unwanted Congested Noise - As mentioned, this decentralized framework is designed to classify any system intrusion as a potential adversarial attack, since it's concealed. This leaves room for an increased rate of false positives *FP*. In our thesis, we have not accounted for regular users *sniffing out* our tokens and accessing the honeypot node, with no intentional adversarial attack in mind. In order to infer that an attack is preemptive, we need to be detect an *adversarial signature*, such as a set of behaviors, anomalies or sequence of events that would indicate an adversarial attack has occurred and classify it as malicious or normal. We have attempted to account for this limitation in our thesis, by utilizing tools that validate actions as *adversarial* or *pre-adversarial* by monitoring whether the attacker violate policies and action specific protocols, which are then compared against a *white-list*. For this very purpose, we are utilizing tools, such *Sysdig* for data capture, *Falco* data control, *Samba* data storage and *Kibani* for data analysis. The latter four are designed to work in synchronization with each other.

Wasted System Resources - Our decentralized defense framework focuses on deploying N honeypots in a structured topology for the adversary to interact with. The system's success depends on the number of honeypot nodes deployed in the environment, as more honeypots create more uncertainty for the attacker. A potential adversary is unaware that none of these deployed honeypots contain the actual target model T , only *decoyed* replicas of the model T_D . The only disadvantage is that the attacker suspects that he is being deceived, and No legitimate classification model exists, to exploit. The latter would likely lead the attacker to a) abort the attack session, b) hijack one of the honeypot nodes. Let it be noted, that this framework is potentially wasteful in terms of infrastructure resources, especially in the case of using high-interaction honeypots, which simulate a real computing environment.

4.3 Design Decisions

Using High-Interaction Honeypots - as mentioned in section 2.4.5, *high-interaction honeypots* (HIHP) simulate an actual and full system for an adversary to attack. The latter includes the os, its applications, input/output and services. But simulating a live system make it time consuming and complex to build, if not virtualized using a VM. a synthetic environment might make it sound like an attractive characteristic, but these type of honeypots utilize more resources than its relative honeypots, exhausting the infrastructure. This imposes a high level of risk on the rest of the host environment when deployed, should it become compromised. Despite all the latter, we found HIHP an optimal choice, for a variety of reasons. For one, the quality of data it can collect is far more extensive and detailed than that of low and medium interaction honeypots, which is limited. Also, it allows the defender to discover unknown adversarial strategies and exposed system vulnerabilities. The latter is not possible in low and medium interaction types since they only simulate and give limited access to the OS. These type of systems are not suited in our case.

Decentralized instead of a Distributed System - Generally, a *decentralized* system is one where the individual nodes are connected to its peers in the network, while a *distributed* system has the nodes distributing work to the sub-nodes. We designed our defense framework to be decentralized for obvious security concerns. Firstly, we wanted the honeypot system and the decoy within to exist as a copy on every nodes interconnected in the system, independently. Secondly, should the adversary's activities within the honeypot become anomalous and illegitimate. The node has a message-passing protocol to send a distress call to the nearest neighboring nodes on network cluster route. These peer nodes will intercept the distress message, verify the certificate sender distress using the public key, then add the IP adversarial information to the central database, and notify other neighboring adversarial honeypot nodes. Memory-sharing used in distributed system was not a benefit we sought in our system, since for privacy concerns we did not want any internal honeypot information to be shared between different honeypots.

Use of the Sysdig-Falco-Samba-Kibana Framework - These 4 tools were selected for data capture, control, storage and analysis restrictively. Generally, *Sysdig* is used for which can save and capture linux machine system state and activity within the machine. *Falco* detects anomalous behavior and their arguments that occurs within the system. *Samba*, an open source log recorder that records the the anomalous events. *Kibana*, a browser based monitor and used for analysis. The latter four tools works efficiently

with each other, using one without the other creates extra and unnecessary work during set-up.

Utilizing Honeybits - this auxiliary tool is used to generate *deception credentials*, used to enhance the effectiveness of the adversarial honeypot defense system. It works by creating breadcrumbs and digital items on production servers to attract the attackers towards a desired honeypot or trap. It will create tokens pertaining to neural networks and machine learning such as data scientist comments, back-up files and data files, etc. All of which are attractive to an adversary. The reason for using them is because they provide a great degree of freedom by allowing us to use the violated integrity of an artificial item to monitor malicious access and signal a compromise. Any item, whether a string, file or email can be made into an token. It is worth noting that the more you plant false or misleading information in response to the post-compromise techniques, the greater the chance of catching the attackers [14].

Black-Box Model instead of Blind model - it has been clearly shown that an adversary can thwart defenses in both context settings, black-box and blind models, as seen in [15] [28] [7] in Chapter 2. The Blind model is more constrained and possesses a very limited (small) set of exposed knowledge to the adversary. But In order for the adversary generate adversarial examples, he must query the *Oracle* O and craft his training-set $D_{training}$ to train his substitute classifier F and transfer the examples to a target classifier T_{Target} to distort the classification of the output labels, as explained in Chapter 2 (section 2.3.2). Having the capability of querying the oracle is only available in a Black-box setting, making it imperative that we establish the attack setting in that matter. For the adversary to be successful in his attack, we must consider the adversary to be at least *weak* - with limited knowledge overall. The adversary only possess limited information because he only observe how the model labels inputs and outputs are produced, with little knowledge of the classifier itself, which is what we assume the attacker has in his disposal in a black-box setting.

Optimization Problem - part of what attracts the adversary to our net of honeypots is the notion of querying the *Oracle* O of the target learning system T_{Target} for input/output pairs (\vec{x}, \vec{y}) to build the substitute model F and then generate the adversarial examples. Generating these examples incurs an expensive cost, seen in the hard optimization problem in section 2.2.5. Solving this hard convex optimization problem is computationally intensive. Ideally, we decided to utilize the same optimization problem

in order to engage the adversary for duration of the attack session in order to collect valuable intelligence. However, the only difference here is that the target learning system the adversary is attacking T_{Target} is a decoy and was specifically designed to attract the adversary to the honeypot, cloaked as a trap.

Scale-out instead of Scale-up - the framework nodes communicate with other peers in the decentralized network, as well as the central Samba database. Scalability in this context, depends on the of decoys we wish to add to the decentralized network to reinforce our defense against adversarial examples. Adding more nodes lowers the probability that an adversary will interact with the actual target model, and lowering the number of nodes increases that same probability. An Adversarial Honeypot node will store a moderate amount information, since high-interaction honeypot only records/stores essential information about the session and store the bulk of the attack data in the central database. Some of the extra data/information stored inside the adversarial honeypot nodes is the Peer-To-Peer authentication key, the software to run Falco and Sysdig, and private key. which is not modest to say the least. The scaling-out model, done *horizontally* entails increasing elasticity by adding more system resources to the existing decentralized framework and increasing it in size by adding more honeypot nodes. Scaling-out becomes a problem if no attackers decide to find and exploit the decoy DNN model T_{decoy} , as it would be exhaustive in resources. However, In the long run, this might be a better decision, since adversarial attacks are projected to increase over time. Scaling-up, done *vertically*, would be more costly since additional resources need to be added to th existing honeypot VM node, increasing availability. Although, this would be unnecessary, unless of course the adversarial threats and the examples become more sophisticated and potent with every new attack. However, this means increased maintenance, costs and research with every new discovered threat, making management and monitoring very complex.

Using Public-Private Key Management - utilizes an *asymmetric key* algorithm to secure intercommunication between two neighboring nodes, *Sender* and *Receiver*. Each node has a key pair, a *Public Encrypting Key* to *Sign* and a *Private Decryption Key* to *Decrypt* communication messages, known only to the node. To combat adversarial attackers from exploiting and hijacking the nodes, we instantiate honeypot intercommunication messages and distress calls to be sent and delivered at any instance of danger. This, as a security measure, to indicate that a node may have become compromised, gone rogue, and prevent any launch of a full DDoS attack on the system. Also, with each

message sent, an authentication message is relayed back from one node to another to insure the sender/receiver nodes meets the required security criteria and is not captured. The neighboring adversarial nodes connected to the primary node vote on whether or not to keep the adversarial node in the network, if it becomes unresponsive to the pulse messages send or does not *verify* it. Each control message will be signed with an authentication token (Private Key), where a Public Key that both nodes will agree on will then be exchanged between them. Generating these Public/Private Keys could be a factor that increases cost should the system scale-out, this something that must be discussed later in this chapter.

4.4 The Threat model

4.4.1 The Adversary

is it targeted or misclassification
 what is being targeted
 how will that manifest

4.4.2 Attacker Capabilities

Each honeypot node in the decentralized defense framework contains a decoy target model T_{Decoy} , presented to the adversary as the legitimate target model. Here, an *Oracle* O represents the means for the adversary to observe the current state of the DNN classifier learning by observing how a target model T_{Target} handles the testing sample set (x', y') . In our attack environment, querying the *Oracle* O with queries $q = \{q_1, q_2, q_3, \dots, q_n\}$ is the exclusive and only capability an adversary possesses for learning about the target model and collecting his synthetic dataset S_0 to build and train his DNN substitute model S .

The adversary can create a small synthetic set of adversarial training samples S_o with output label y' for any input x' by sending $q_n > 1$ queries to the *Oracle* O . The output label y' recurred is the result of assigning the highest probability assigned a label y' , which corresponds to a given x' . This the only capability that the attacker for learning about presumed target model T_{Target} through its *Oracle* O . The attacker has virtually no information about the DNN internal details. The adversary is restrained by the same restrictions a regular user querying the *Oracle* O has. latter is something an adversary should adhere to make his querying attempts seem harmless, while engaging the decoy model within the adversarial honeypot. Finally, we anticipate that the adversary will

not restricted himself to one honeypot, and will likely connect to multiple nodes and DNN classifiers from the same connection for purposes of parallelizing synthetic data collection. This should trigger an alarm within our framework, indicating multiple access and is something abnormal is occurring.

4.4.3 Attack Setting

As mentioned in the assumption part of this chapter (section 4.2), our envisioned profile for the adversary targeting our Black-Box learning system does not possess any internal knowledge regarding the core functional components of the target model T_{Target} DNN. This restriction entails no access to model's DNN architecture, model hyper-parameters, learning rate, etc. We have already established that an adversary can prepare for an attack by simply monitoring target model T_{Target} through its *Oracle* O and use the labels to replicate and train an approximative architecture.

The ad-hoc approach at the adversary's disposal is that he can learn the corresponding labels by observing how the target model T_{Target} classifies them during the testing phase. The adversary can then build his own substitute training model F and use this substitute model F in conjunction with synthetic labels S_o to generate adversarial examples propped against the substitute classifier, which the attacker has access to. Even if the substitute model S and target model T_{Target} are different in architecture, the adversarial examples x^* generated for one can still tarnish the other if transferred using *adversarial transferability*. Since the adversarial examples between both models are only separated by added tiny noise ϵ , the examples look similar in appearance. The latter is true even if both models, original T_{target} and substitute model F , differ in architecture and training data. As long both models have the same purpose and model type. Although the *Adversarial transferability* phenomena is discouraging, but alone it is advantageous for the adversarial attackers to launch targeted attacks, with little or no constraint on their attack blueprint. Adversarial transferability eventually becomes a serious concern because attacks will grow in sophistication and potency over time. It's challenging to design a model that can generalize against more advanced attacks, if not all. Also, it's difficult to dismantle and reverse-engineer how these attacks propagate and cause harm, since no tools exist to expedite the process to learn from the attack in time.

4.4.4 Exploited Vulnerability

4.4.5 Attack Strategy

4.5 Decoy DNN Model

4.6 Adversarial Honeypot Network Overview

4.7 Individual Honeypot Topology

4.7.1 Purpose

4.7.2 Architecture and Topology

4.7.3 Training, Testing and Validation

4.8 Attracting The Adversary

4.8.1 Adversarial Tokens

4.8.2 Weak TCP/IP ports

4.8.3 Decoy Target Model

4.9 Detecting Malicious Behavior

4.10 Monitoring the Adversary

4.11 Launching The Attack

4.12 Defending Against Attack

4.13 Deployment

4.14 Scalability

4.15 Security

4.16 Significance and Novelty

Prevent Target-Substitute-Example-Transferability via Deception - the ingenuity of our approach lies in how we solve the problem of blocking *adversarial transferability* from occurring. Our decentralized proxy defense framework behaves as an *extra* layer of security for learning systems within a black-box setting. The latter is achieved by *deceiving* and *exploiting* the adversarial attacker and not blocking the adversary from querying the *Oracle* O of the true target model T_{Target} . A series of deception methods are deployed to *lure*, *trap* and *exploit* the adversary. To be concise, once an adversary is contained within the honeypot, and while under the notion that contained environment is real, our system already anticipates an attack using adversarial examples. The unsuspecting attacker generates his adversarial samples \vec{x}^* from observing how the *Oracle* O labels input and output (x_i, y_i) to build his substitute model F . The adversary, assuming his attempts are successful has been led into a trap. The adversarial examples generated are from the decoy substitute model F not usable *per-se*, in the sense that they are only useful against the deployed decoy model T_{Decoy} . Even though the non-linear

optimization problem of generating suitable perturbations is solved, and the adversarial examples are generated and ready for insertion it has been done so as wasted labor for a decoy.

Fail-Safe to supplement *less-than-secure* Defense Techniques - literature regarding adversarial defense techniques such as Adversarial Training [12] and Defensive Distillation [7] [28] have been considered to be insufficient in their capacity to bar an adversary from using adversarial examples to influence the classification model. For instance, even under the *blind model* where the adversary’s knowledge on the DNN model is virtually absent, the attacker was still about to mount a successful attack, as seen in [15]. In our framework, we designed a helper layer to *screen* and filter potential adversaries who are looking to exploit the model. It works by augmenting existing security measures by using methods of deceit and deception to entice the advancing attacker by luring him away from the target model towards a decoy model T . Our method does not, in anyway, increase the complexity of deploying security sensitive machine learning systems in the real-world. Simply put, it works by obstructing the efforts and fazing adversaries, by reciprocating the adversary efforts with deception.

Adversarial Information Reconnaissance - we decided to use high-interaction honeypots, not solely for the purpose of *fooling* or baiting the adversary, but for more practical reasons as well, such as watching adversarial behavior. Some types of honeypots possess the ability to monitor the adversary’s malicious activities within the honeypot, and record information about the exploitation session. The latter was the reason for selecting *high-interaction* honeypots. This data can potentially used to analyze the adversary’s motives, as well as trends, new tool being used adversaries, and any personal motives. This can be immensely useful if the adversary decides to return in the future. Honeypots as a tactic have never been used before in the fight against adversarial examples in Black-box systems, until now.

Includes a 3-tier Deception Mechanism - Our method of defense focuses on deception as a method to prevent *transferability* from occurring, this 3-tier deception system helps us to do that:

- *Adversarial Honey-Tokens (between-attacker-and-network)* - These adversarial tokens will be uniquely generated *fictional* words or data records deployed on production servers, which will be used to lure the attacker to the honeypots running on the servers. These tokens do not normally appear in normal network traffic,

which is exactly why it will seem alluring to the attacker. These tools will allow the defender to attract the adversary towards a prepared trap. We will use an extended version honeybits [14] repository to generate tokens pertaining to the DNN classifier and use the Linux based tool *Auditd* to monitor token access.

- *Weak point of entry TCP/IP port (between-attacker-and-honeypot)* - We assume the attacker will find and interact with our desired honeypot VM if we leave a vulnerable way in. This will be done by keeping a TCP/IP port open for the attacker to listen to and interact with the honeypot. The adversary and honeypot will initiate the 3-way handshake to establish the connection (or perhaps not finishing it). We assume the attacker is using port scanning software such as *Nmap* or *Nessus* tools.
- *decoy DNN model - (between-attacker-and-environment)*

Chapter 5

Implementation of Adversarial HoneyTokens Component

In this chapter, we provide a detailed breakdown of the adversarial honeytoken implementation mentioned in Chapter 4. **This component was built as an extension to the pre-existing *honeybits* github repository** provided to us for use from [14]. Firstly, we provide a background to these tokens, as well as explain the nature and purpose of the honeytoken bits. Then, we provide an outline of the individual project components and its hierarchical structure. The rest of this chapter focuses on software architecture and design, functional features, as well as usage, deployment scenarios and strategies. We then end this chapter with a highlight of the external dependencies which our extension - *Adversarial Honeytokens* and its predecessor depends on, integration, and benefits.

To get a deep and through understanding of the honeybits source code, as well as the contributions made by the thesis author please see Appendix A in Chapter 7.

5.1 Background

The *honeybits* provided in [14] is a simple tool developed in 2017 by the software developer named *Adel Karimi*, with alias *0x4d31*. The initial purpose behind the design of this tool, according to the developer in [14] was in verbatim to: *"enhance the effectiveness of your traps by spreading breadcrumbs & honeytokens across your systems to lure the attacker toward your honeypots"*. From the view of the defender, one can see the reasoning behind designing the initial version of this tool - to solve the problem that plagues most networking scanning tools used by adversaries to survey and monitor the

network, which is that it did not effectively filter out unwanted noise from the external environment. This problem meant that adversaries using these network scanners cannot identify the hidden honeypots masqueraded as legitimate targets in the production environments that security personnel wanted the adversaries to fall victim to. Lack of attacker interest was the very reason for the inception of this tool; to enhance the effectiveness of finding honeypots by the adversaries, so they may be lured, deceived, trapped and their methods studied [14].

The honeybit tool can be used for various tasks and purposes. However, its main purpose is to cease the pursuit of adversaries and instead allow them to come to the honeypot independently. It appears the designer of this system intended it to be used as a simple and cost effective deception technique to generate, distribute and plant falsified and bogus information. The latter in response to a risk of compromise or intrusion on any of the production environments. This tool can be quite useful if used strategically, especially since it can be said that the more falsified information is planted, the greater chance an attacker, internal or external, will be deceived by this information and fall victim to our deception [14]. The *honeybits* tool enables the defender to automate the design and creation of electronic decoys are hard to distinguish from real objects. These tokens behave just like honeypots, in the sense they not useful unless interact with, with the subject interacting with them being someone with malicious intent or with unauthorized access, such as an adversary. As mentioned in section 2.4.2, these *digital* pieces of information can take any form or structure the creator chooses. But regardless of what the true form of the tokens are, they're fake, bogus and of no real value. For example, the token could be that of a *Canadian Social Insurance Number* (SIN), this number could be embedded inside a database system as a honeytokens, if the number is accessed, we know that someone is attempting to violate the integrity of the system. It is worth noting that these tokens must seem authentic and real, to be attractive to the attacker. An Intrusion-Detection-System (IDS) could be used to detect when the digital token is unauthorizedly accessed, and the simply "call home".

The novelty with this bait distribution software is in the wide variety of tokens it is able to generate and customize, as well as the embedding options associated with them. This software is able to generate tokens pertaining to *ssh*, *ftp*, *rsync*, *ftp*, *mysql*, *wget*, and *aws*, as well as enable the use of custom tokens and templates.

The honeybit software was designed with a set of features and functionality in mind. Initially, the honeybits breadcrumb software was used to generate, distribute and plant: 1) Fake Bash history commands, 2) Fake AWS credentials, 3) Fake configuration, backup and connection files, 4) Fake entries in host and ARP table, 5) Fake browser history,

bookmarks and saved passwords, and 6) Fake registry keys.

A portion of my thesis utilizes these tokens as part of a proposed auxiliary defense method, used to enhance the effectiveness of my adversarial honeypot defense system to prevent adversarial transferability from transpiring. We decided to extend the existing API in order to generate custom breadcrumbs and digital tokens on production servers that attract adversaries towards the desired honeypot, and the decoy DNN target model within. Hence, we extended the current architecture to create tokens pertaining to training and testing of the machine learning model, data scientist comments, emails, back-up and data files, etc. These tokens in their variants are fake objects (that look real) which would not normally appear in network traffic, which makes them very attractive to an adversary.

5.2 Project Structure

The honeybits project and by extension the adversarial honeytokens has the following hierarchical structure:

- └── **LICENSE**
- └── **README.md**
- └── **adversarial-honytokens**
- └── **hbconf.yaml**
- └── **honeybits.go**
- └─ contentgen/
- └── **contentgen.go**
- └─ docs/
- └── **honeybits.png**
- └─ template/
- └── **rdpconn**
- └── **trainingdata**
- └── **testingdata**
- └── **txtemail**

5.3 Architecture

Here, we discuss the role and responsibility of each source code file in the project. See Figure 5.1 below for the general architecture of the adversarial honeytokens. In Figure

5.1 we can see how the different components interact with each other to generate the adversarial honeytokens:

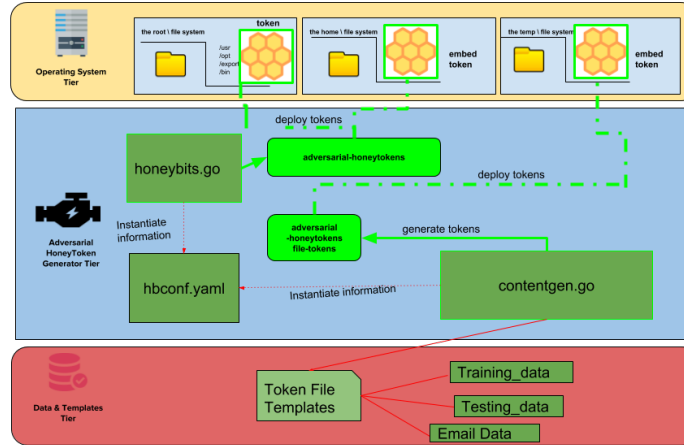


Figure 5.1: Adversarial HoneyToken Architecture

- **contentgen/contentgen.go** - this source code file is responsible for instantiating the IP address other and information from .yaml configuration file. Also, it works on collecting formatted content from the templates to generates the file tokens, such as the *txtmail* and *trainingdata* token. Please see Appendix A in Chapter 7 for source code.
- **template /txtemail** - is the sample template used to generate the txtmail file token to entice the attacker. It details the interaction between the security system administrator and resident data scientist. Please see Appendix A in Chapter 7 for source code.
- **template /trainingdata** - is the sample template used to generate the trainingdata file token to entice the attacker. Please see Appendix A in Chapter 7 for source code.
- **template /testingdata** - is the sample template used to generate the testingdata file token to entice the attacker. Please see Appendix A in Chapter 7 for source code.

- **hbconf.yaml** - this .yaml (*Yet-Another-Markup-File*) markup language file is responsible for the configuration settings of the adversarial honeytokens. Through this file, the defender can customize and add file tokens, as well as the individual network tokens of the commands, such as *ssh*, *scp*, *ftp*, *wget*, *aws* etc. Also, we can customize the individual custom tokens pertaining to the DNN design, configuration, testing and training. It is also possible to customize the paths of where the tokens will be generated and embedded for the attacker to find. Please see Appendix A in Chapter 7 for source code.
- **honeybits.go** - used to create the file, general and custom honeytokens specified in the .yaml configuration file. It is responsible for collecting all the formatted information provided in the .yaml file, creating the tokens, and deploying them inside their respective embedding locations. Please see Appendix A in Chapter 7 for source code.

5.4 Features

This section is divided as follows, existing features originally implemented by the developer (Adel *0x4D31* Karimi) in [14], and features and additions added by the thesis author (Fadi Younis):

- **Existing Features:**
 - i. Creating honeytokens that can be monitored using Audit GO.
 - ii. Template based generator for honeyfiles.
 - iii. Insert honeybits into `/etc/hosts`.
 - iv. Insert different honeybits into "bash-history", including the following sample commands.
 - v. Modifying the code base to allow generation of honeytokens related to machine learning model configuration, testing, training and deployment, that would seem attractive to an adversary.
- **Added and Extended Features:**
 - i. Design and deployment of the custom adversarial honeytokens related to the deep learning model, inserted into the bash history file of the operating system. Tokens, such as those related to the deployment, configuration of the model, as well as testing and training.

- ii. Design and deployment of adversarial file tokens, such as training data and email.
- iii. Design and writing Linux auditd rules for monitoring, accessing, and accounting of the adversarial tokens.
- iv. Deployment of the adversarial tokens inside the Linux container for purpose of running the application in a controlled and monitored environment, to be deployed anywhere within the operating system.

5.5 Functionality

We extended the honeybit token generator in [14] to create the *adversarial honeytokens* generator, which acts as an automatic monitoring system that generates adversarial deep learning related tokens. It is composed of several components and processes, as seen in Figure 5.1 above. In order to understand how the system functions, one must have an understanding of the individual operative components and processes. The following points offer an insight into how the system functions used to create token and decoy digital information to bait the adversary.

Baiting the Attacker - In order for the digital tokens generated by the application to bait the attacker successfully they should have the following properties: 1) be simple enough to be generated by the adversarial honeytokens application, 2) difficult to be identified and flagged as bait token by the adversary, 3) pragmatic to pass itself as a factual object, which makes it difficult for the adversary to discern it from other legitimate digital items. The purpose of these monitored (and falsified) resources is to persuade and lure the adversary away from the target DNN model T_{target} , and bait him to instead direct his attack efforts towards a decoy model T_{decoy} residing within the honeypot trap. The goal here is to allow the adversary's malicious behavior to compromise the hoaxed model, preventing the actual adversarial transferability of the T_{target} model from occurring, and forcing the attacker to reveal his strategies, in a controlled environment. The biggest challenge associated with designing these tokens is adequate camouflaging to mimic realism, to prevent being detected and uncloaked by the adversary.

Adversarial Token Configuration - the configuration of the adversarial honeypot generator occurs within the .yaml markup file (hbconf.yaml). Here, the administrator sets the honeypot decoy IP address, deployment paths, and content format. The configuration file, through the path variables, set where the tokens will be leaked inside the

operating system, offering by that a large degree of freedom. Also, the administrator can customize the individual file tokens, as well as the general honeytokens and the adversarial machine learning tokens added. As mentioned, this file allows the building of several types of tokens. The first type of tokens are the *honeyfiles*, which includes *txtmail*, *trainingdata*, and *testingdata*, these type of tokens are text based and derive their formatted content from the template files stored in the templates folder. The second type of tokens include network honeybits, which include fake records deployed inside the UNIX configuration file or any arbitrary folder. The latter include general type tokens such *ssh*, *wget*, *ftp*, *aws*, *etc*, These tokens usually consist of an IP, Password, Port, and other arguments. The third type of tokens deployed are the custom honeytokens which are deployed in the bash history; these tokens are much more interesting since they take any structure or format the defender desires.

Adversarial Token Generation - through the extended adversarial token framework we compile the tokens using *go build* command. The following are only some of the tokens that can be generated using the adversarial honeytokens framework:

- **General Honeybit Tokens**

- *ssh token*
- *host configuration token*
- *ftp token*
- *scp token*
- *rsync token*
- *sql token*
- *aws token*

- **File Tokens**

- *txtmail token*
- *training data token*
- *testing data token*
- *data scientist comments tokens*

- **Custom Honeybit Tokens**

- *ssh password token*

- *training data copy token*
- *testing data copy token*
- *start cluster node token*
- *prepare python DNN model token*
- *train python DNN model token*
- *test python DNN model token*
- *deploy python DNN model token*

Token Leakage - the most dominant feature of the adversarial honeypot generator is its ability to inconspicuously implant artificial digital data (credentials, files or commands), etc) into the productions server's file system. The embedding location can be set inside the .yaml configuration file (hbconf.yaml) using the PATHS: *bashhistory*, *awsconf*, *awscred* and *hosts*. After the defender compiles and builds the adversarial tokens they are stealthily deployed at set path locations within the designated production server's operating system. There, the tokens reside until they are found and accessed by the adversary. The Docker container at this point records intelligence on the attacker's interaction with the token. See Figure 5.2 below for an illustration of the adversarial token leakage into the adjacent production systems.

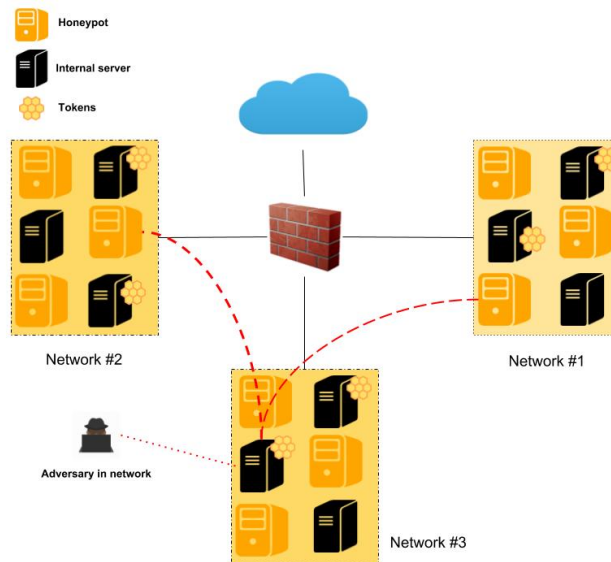


Figure 5.2: Adversarial Token Leakage

Audit and Control Rules - the auditd daemon was used to monitor activities within the docker container. It can be used to monitor anything from system calls to wide network traffic. Among the many capabilities this tool has, it can do the following: 1) see who accessed/changed a particular file within the file system, 2) monitor system calls and functions, 3) detect anomalies such as running and crashing processes, 4) set trip wires for intrusion detection, 5) record any commands entered. However, it used for a specific task in mind, which was to monitor access to adversarial honeytokens deployed in a specific location within the file system. Configuration and customization of the daemon is done through the configuration daemon file *audit.conf*, while the control file *audit.rules* controls customization of the monitoring rules.

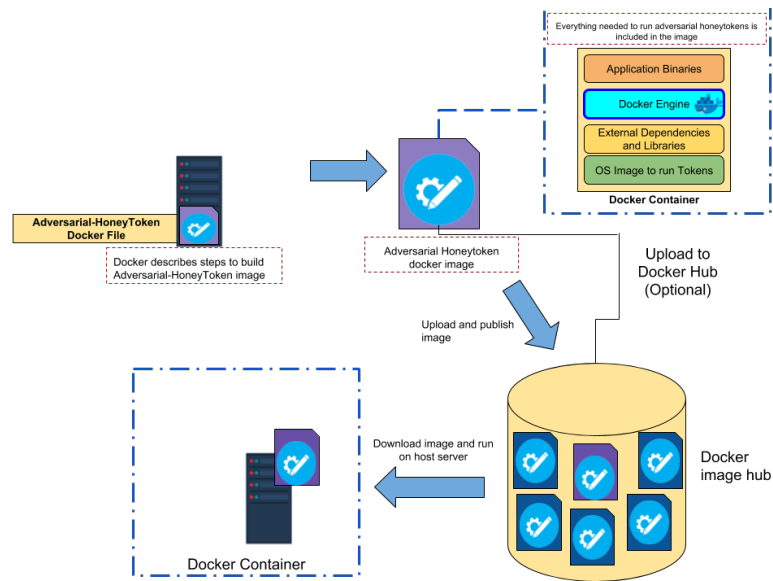


Figure 5.3: Dockerize Adversarial HoneyToken Application

Docker to Monitor the Adversary Access - Docker was selected since it provides a free and practical way to contain application processes and simulate file system isolation, where the adversarial tokens application image will be run. In our defense framework, the numerous production servers not open to the public domain will be reserved for adversarial research to capture intelligence and analyze attacks. They will open via an exposed TCP/IP port open to the public, with weak access points. The docker container will act as the *sandbox*, acting as entire layer to envelop the honeypot application image. Using the insight gained from the adversaries later lured to the honeypots will be used study emergent adversarial strategies, input perturbations and discovering techniques

used by adversaries in their exploits. Docker will create a new container object for each new incoming connects and set up a *barrier* represented as the sandbox. An unsuspecting attacker that connects to the container and finds the tokens is presumably lured to the honeypot containing the decoy DNN model. If the adversary decides to leave, he is already keyed to that particular container using his IP address, which connects him to the same container if he decides to disconnect and then reconnect.

5.6 Usage

In this section, we present three attack and exploitation scenarios where the adversarial honeytokens will prove themselves to be useful. In each of these speculative situations, we present the setting in which the attack occurs, the objectives achieved by deploying the tokens, the environment set-up required to deploy the tokens, the tokens generated, as well as token generation, building and compilation. Then we delve into how the audit rules are set to monitor token access, as well as what results we achieve from deploying these tokens. Consider the following scenarios.

scenario 1 - Luring Away an Unwanted Adversary

- **Setting**

- Security informatics company *CyberLink* has deployed a deep learning binary classification system to classify network traffic data. CyberLink has recently been a victim of several cyber attacks targeting their classification system. System Administrator Jane Doe has decided to deploy a high-interaction honeypot embedded with a decoy DNN model, in an attempt to divert the attacker away from the target model using custom-designed fake digital tokens called Adversarial HoneyTokens. These tokens contain information to entice the adversary and lure him away from the target model towards the decoy within the honeypot. Once the honeybits are installed inside a Docker container along with auditd scripts to monitor token access. The Docker software creates a new container for each connection. Finding what he, or she, thinks is an easy target, the attacker let's call him John Doe accesses the tokens. the falsified information inside the token lured the attacker to a production system embedded with the decoy DNN model. Figure 5.4 below illustrates the typical adversary interaction with the honeytokens manifested in scenario 1 (*Luring Away an Unwanted Adversary*).

- **Objectives**

- generate falsified digital tokens and *breadcrumbs* that appear tempting to an adversary. Then package and deploy the tokens within the file system to be discovered and mislead the adversary.
- use the tokens to keep attacker at bay and to protect the deployed classification system from exploitation.
- prepare tokens that give the defender the freedom to design, deploy and monitor tokens that signal an exploit once an adversary accesses them.

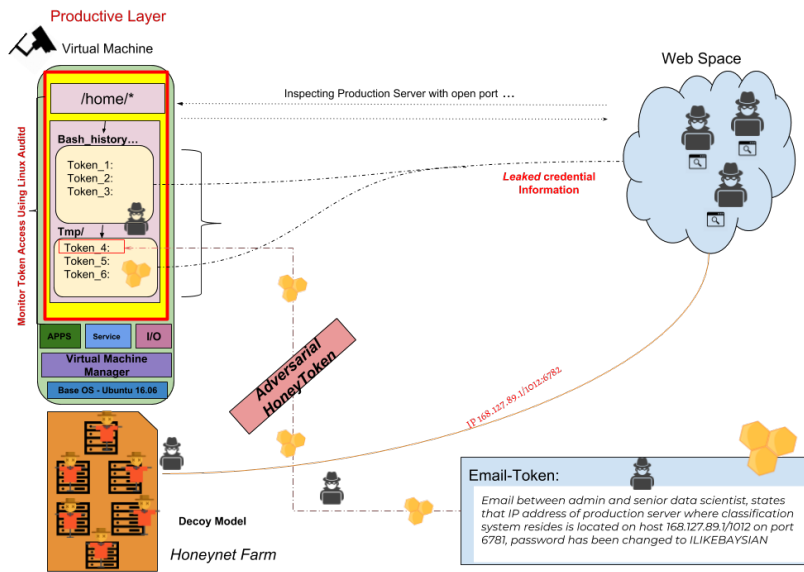


Figure 5.4: Scenario1 - Luring Away Attacker from Target Model

• **Tools Used**

- Adversarial HoneyTokens generator
- VM Player
- GoEnv Virtual Environment
- Linux OS Auditd Daemon
- Docker

• **Environment Set-up** - In order to activate the GO virtual environment for the GO programming language using compiler version 1.7, do the following:

- Navigate to */adversarial-HoneyTokens* folder inside main project folder.

- Select the compiler version and activate the virtual Go environment using the following command (see attachment 1.1 in Appendix A):

```
goenv --go-version=1.7
```

- ***Tokens Used***

- ***Email Token*** - Deploy the sample email token file, sent between the administrator/senior data scientist indicating that the IP address of production server where classification system resides is located on host 168.127.89.1/1012 on port 6781, and password has been changed to I-LIKE-BAYSIAN. The token is embedded and deployed within */tmp* folder.
- ***Password Change Token*** - password of the production server is changed I-LIKE-BAYSIAN
- ***Training/Testing Data Tokens*** - Deploy data training and testing files embedded within */tmp* directory.

- ***Building the Tokens***

- In order to build these adversarial honeytokens before compilation, run the following command:

```
go build
```

- ***Setting Audit Rules***

- Email Token**

- Monitoring a path by putting a *watch* monitor for read/write/execute/attribute change. We monitor the following path: */tmp*

TO SET RULE:

```
sudo auditctl -a exit,always -F path=/tmp -F perm=rwx
```

Password Change Token

- Monitoring a path by putting a watch for read/write/execute/attribute change:
- We monitor the following paths: */home/test*

TO SET RULE:

```
sudo auditctl -a exit,always -F path=/home/test -F perm=rwx
```

Training/Testing Data Tokens

- Monitoring a path by putting a watch for read/write/execute/attribute change:
- We monitor the following paths: */home/test*

TO SET RULE:

```
sudo auditctl -a exit,always -F path=/home/test -F perm=rwx
```

- ***Building and Compiling***

- In order to compile the adversarial honeytokens, run the following command (see attachment 1.2 in Appendix A):

```
./adversarial-tokens
```

- If *successful*, the tokens created will be stored *bashhistory*, *tmp*, etc.
- To see the embedded tokens:
 - navigate to the *tmp* folder and display hidden files for email token using *ls -a* command.
 - navigate to */home/test* and open the *bashhistory* file for training/testing tokens, see attachment 1.3, 1.4, 1.5 in Appendix A.

- ***Monitor Token Access***

- **Inside */home/test***

- **Find file access by tracing all recent activities within folder */home/test***

```
ausearch -f /home/test
```

- **Search information within the logs captured**

```
sudo autrace /home/test  
sudo ausearch -i -p 1070
```

- **Inside */tmp/***

- **Find file access by tracing all recent activities within folder */tmp/***

```
ausearch -f /tmp
```

- **Display information logged captured**

```
sudo ausearch -p 1070 | aureport
```

- Search information logged captured

```
sudo  autrace  /home/test  
sudo  ausearch -i -p 1076
```

- Display the ID of the user who access the path and files for a suspected user

```
sudo  auditctl -a exit ,always -F arch=x86_64 -S open -F auid=1000
```

- *Accomplishments*

- Generate adversarial tokens and deploy them within a part in the production server file system, where the attacker is most likely to search for.
- Customize the adversarial tokens for neural net, including tokens pertaining testing/training and configuration.
- Monitor token access and manipulation through the auditd monitoring daemon, as well track and record the attackers user information.

scenario 2 - Discourage Future Attacks Cyber Security company *CyberLink* has deployed an multi-class classification system to classify security footage as either incriminating activity or benign. However, recently there has been a slew of adversarial attacks compromising the integrity of their classification systems and exploiting their DNN with adversarial examples that cause mislabeling. As a countermeasure and an attempt to combat and bewilder the adversarial threats, the defender has decided on deploying high-interaction honeypots masqueraded as production systems in the environment. Accordingly, in order to attract the attacker the defender has decided to leak an obscure *SSH* token indicating the deployment and existence of 1000 honeypots deployed within the system. An adversary, knowing the latter, would be discouraged from launching further attacks since the type of deployed honeypot can be used reverse-engineer the attacker's strategies. For an illustration of scenario 2, please refer to Figure 5.5 below

scenario 3 - Apprehend an Internal Adversary Cyber Security company *CyberLink* suspects that an internal malicious attacker is compromising their multi-class classification system. The administrators suspect the attacker is interested in violating the integrity of their classification system. A high-interaction honeypot is deployed in a subnet of the network mimicking a genuine production server host with a decoy DNN model. A Token is inserted into the *bash history* indicating the classification system was recently updated and redeployed into the production environment. The adversary

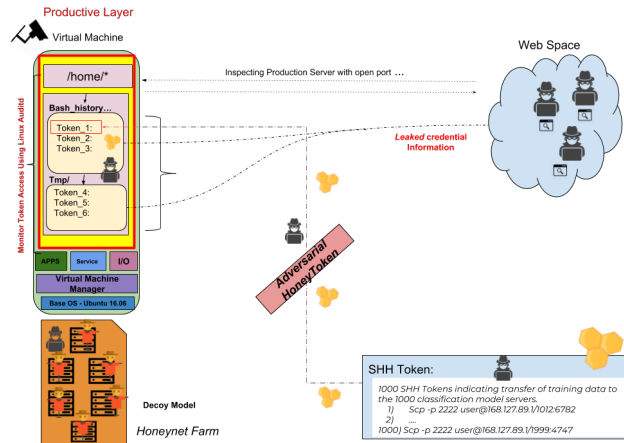


Figure 5.5: Scenario 2 - discourage future attacks

maybe very hard to catch since his activities resemble real traffic. This is why the tokens are deployed in plain sight of the adversary's path on the internal network. The token can be designed to attract the adversary to a newly deployed decoy classifier. For an illustration of scenario 3, please refer to Figure 5.6 below.

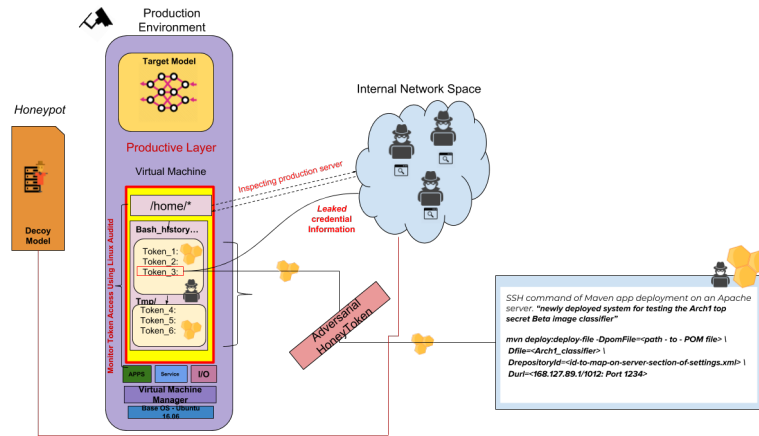


Figure 5.6: Scenario 3 - apprehend an internal adversary

5.7 External Dependencies

The adversarial honeytokens project has the following outside external dependencies:

- **G0 Lang** - an open source programming language [1] which was used as the main development language for building the honeytokens API. Also, the same language

was used to extend the same API to build the adversarial honeytokens.

- **GO-env Virtual Environment** - a Go version management utility software tool [32] which allows the user to setup an isolated Go virtual environment on a per-project basis, per shell basis. It allows the user to install different Go compiler versions, as well as set up isolated Go environment variables, such as ROOT and DEBUG.
- **Viper** - a configuration solution for GO applications [11]. It is designed to work within an application, and can handle all types of configuration needs and formats. Viper can be thought of as a *registry* for all of the GO application's configuration needs. It supports the following features [11]:
 - 1) setting defaults
 - 2) reading from JSON (JavaScript Object Notation), TOML (Tom's Obvious, Minimal Language) , YAML (Yet Another Markup Language), and Java properties
 - 3) config files
 - 4) live watching and re-reading of config files (optional)
 - 5) reading from environment variables
 - 6) reading from remote config systems (etcd and Consul)
 - 7) watching changes
 - 8) reading from command line flags
 - 9) reading from buffer
 - 10) setting explicit values.
- **Crypt** - a configuration library, used to compress, encrypt, and encode encrypted GO application configuration files using a secure public key [4]. It can be thought of as a kind of *key ring*, created from a batch file.
- **Linux Auditd Daemon** - used to monitor security level events in the Linux operating system, it be used for the following tasks [3]:
 - 1) monitor accessed/changed a particular file
 - 2) detect system calls and functions
 - 3) record anomalies, such as running and crashing processes
 - 4) set trip wires for intrusion detection
 - 5) record any commands entered
- **Docker Container Software** - an open source tool designed to simplify the process of creating, deploying, and running applications by using containers [2].

Docker allows a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.

5.8 Integration

The adversarial honeytokens generator, alone, as a separate component, can be used in a variety of different domains and areas, here are just some of them:

- *catching inside/outside adversaries in the act* - outside the scope of adversarial examples and black-box systems, an organization suspecting that an internal malicious adversary compromising their classification system can benefit from using our deception tool. A honeypot can be masqueraded as a production system in one of the network subnets, mimicking a genuine host system deployed with a deep learning decoy classifier. Since the attacker is an unknown insider, he could potentially be difficult to apprehend since his activity signature resembles that of benign network traffic, which might pose as a challenge to differentiate from an actual adversary. A defender can use the token generator to craft digital items potentially attractive to an attacker, that would be placed in plain sight to bait and draw-out the adversary from his hiding place .
- *unanticipated attacks on MLaaS* - we know that self learning systems in the form of a service are vulnerable to well-crafted adversarial attacks. The latter opens the possibilities for the type of aggressive security risks that can target and evade these online services, leaving organizations vulnerable. One way to combat this is with using the adversarial token generator, which counters the adversary's aggression and keeps the attacker at bay using deception, as a method of defense.
- *protecting privatized classification systems vulnerable to attack* - some classification system are publicly available for querying, and instead reside as an hidden internal component of an organization's domain. An adversary haphazardly lurking within the system can still potentially corrupt the deep learning model. However, if the attacker has less than ideal knowledge of the classifier, it can make designing the honeytokens a lot easier since the attacker will have great difficulty identifying the genuine model.

5.9 Benefits

The adversarial honeytokens generator tool has multiple benefits that can significantly improve the security of a classification system, residing in any organization. The greatest yield from its usage lays in the freedom associated with the ability to create, design and deploy these entities. The defender has a great advantage to potentially use this tool with the acquired knowledge of the adversary he might already possess. Knowledge of the adversary's techniques, methods and motives might help us design more sophisticated and pragmatic tokens to bait the adversary. Here are some of the other benefits associated with using this tool:

- this tool provides the ability to generate falsified digital tokens and deploy them within a masqueraded production system, *sandboxed*' within docker container. These tokens are designed to be embedded almost anywhere, but in order for minded adversary to discover them (and be deceived by them) these tokens must be deployed in a location or PATH frequented and known by the adversary.
- honeytokens, if designed strategically, can mislead and keep an attacker at bay. This supplementary tool can potentially be used with other defense to protect classification systems deployed within public and private environments. Some of the conjectural scenarios where a benefit can be seen are mentioned above in section 5.6 (Usage).
- this tool gives the defender ability to hide traps among legitimate files within the production. This affordable and efficient tool enables us to exploit the adversary's strategy, which is based on *trust* - trust that there is no confusion or misdirection by the defender.
- through the use of Docker containers, this allows we can generate, deploy, and monitor access of the adversarial tokens practically on any system, giving this extended framework high portability.

Chapter 6

Conclusion and Future Work

6.1 Summary

6.2 Discussion

6.3 Contributions

6.4 Future Work

6.5 Conclusion

Chapter 7

Appendix A

7.1 Adversarial Honey Source Code

The following section details the source code used to generate the adversarial tokens, along with the extended functionality to extend it to build the adversarial honeytokens. The source code displayed in this section includes: 1) *contentgen.go*, 2) *hbconf.yaml*, 3) *textemail token* and 4) *honeybits.go*.

7.1.1 contentgen.go

```
// Copyright (C) 2017 Adel "0x4D31" Karimi
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is decentralized in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <http://www.gnu.org/licenses/>.

package contentgen
```

```
//imports
//format, Viper repo, I/O, OS
import (
    "fmt"
    "github.com/spf13/viper"
    "io/ioutil"
    "os"
)
\
//READ TEMPLATE
//Opens the file template, display error in stderr if no file
//Else store file in string variable
func readtemplate(tp *string, fp string) {
    if fi, err := ioutil.ReadFile(fp); err != nil {
        os.Stderr.WriteString(fmt.Sprintf("Error: %s\n", err.Error()))
    } else {
        *tp = string(fi)
    }
}

//GENERATE Text using configuration viper file, and 2 strings
func Textgen(conf *viper.Viper, ctype string, ctemp string) string {
    //initialize address of the honeypot from the configuration file
    addr := conf.GetString("honeypot.addr")
    //initialize honeypot data to " "
    data := ""
    //initialize template data to " "
    template := ""
    //Use Default template
    t := &template

    //switch case statement for template
    switch ctype {
    //REMOTE DESKTOP CONNECTION CASE
    case "rdpconn":
```

```
if ctemp == "config" {
//Get the template data from RDP template
*t = conf.GetString("contentgen.rdpconn.template")
} else {
readtemplate(t, ctemp)
}
//SET INFORMATION
if ap := &addr; conf.IsSet("contentgen.rdpconn.server") {
*ap = conf.GetString("contentgen.rdpconn.server")
}
p := &data
//PRINT INFORMATION
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.rdpconn.user"),
//TEXT EMAIL CASE
case "txtemail":
//Get the template data from txtemail template
if ctemp == "config" {
*t = conf.GetString("contentgen.txtemail.template")
} else {
readtemplate(t, ctemp)
}
//SET INFORMATION
if ap := &addr; conf.IsSet("contentgen.txtemail.server") {
*ap = conf.GetString("contentgen.txtemail.server")
}
p := &data
//PRINT INFORMATION
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.txtemail.user"),

//TESTING
case "testing":
//Get the template data from testing template
if ctemp == "config" {
*t = conf.GetString("contentgen.testing.template")
} else {
readtemplate(t, ctemp)
```

```
}
//SET INFORMATION
if ap := &addr; conf.IsSet("contentgen.testing.server") {
*ap = conf.GetString("contentgen.testing.server")
}
p := &data
//PRINT INFORMATION
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.testing.path"),
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.testing.date"),
//TRAINING
case "training":
//Get the template data from testing template
if ctemp == "training" {
*t = conf.GetString("contentgen.training.template")
} else {
readtemplate(t, ctemp)
}
//SET INFORMATION
if ap := &addr; conf.IsSet("contentgen.training.server") {
*ap = conf.GetString("contentgen.training.server")
}
p := &data
//PRINT INFORMATION
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.training.path"),
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.training.date"),
//DEFAULT CASE – HELLO WORLD
default:
p := &data
*p = "Hello World!"
}
//return the data
return data
}
```

7.1.2 txtemail template

From: admin <adel@example.com>

Subject: Re: password change
Date: April 18th, 2017 at 21:59:15 GMT+11
To: JOHN DOE <JOHN.DOE@example.com>
Cc: security <security@example.com>

Hi,

Ah, sorry I forgot to send you the new address: `http://%s`
I also reset your password (user: `%s`) to the default pass: `%s`

Please set the MFA (multi-factor authentication) ASAP.

Cheers,
Adel

On April 18th 2017, at 9:57 pm, admin <dave.cohen@example.com> wrote:

Hi admin,

I just wanted to login to the Monitoring system, but I get 404 error. Could

Thanks
Dave

The information contained in this email and any attachments is confidential

7.1.3 hbconf.yaml

```
#PATHS
path:
  bashhistory: /home/test/.bash_history
  awsconf: /home/test/aws/config
  awscred: /home/test/aws/credentials
  hosts: /etc/hosts

#WHAT FILES TO USE
randomline:
```

```
bashhistory: true
confile: true
```

~~#~~HONEYPOT ADDRESS

```
honeypot:
addr: 192.168.1.66
```

~~#~~FAKE FILES

```
honeyfile:
enabled: true
monitor: auditd # Options: go-audit, auditd, none
goaudit-conf: /etc/go-audit.yaml # Only if you use go-audit
traps:
```

```
# Format: - file_path:content_type:template
```

```
## content_type: rdpconn, txtemail,
```

```
## template: config (read from config file: contentgen.xxx.template), templ
```

```
- /tmp/test.rdp:rdpconn:config
```

```
- /tmp/email.txt:txtemail:template/txtemail
```

```
- /tmp/testing:testing-data:template/testing
```

```
- /tmp/training:training-data:template/training
```

```
# Content generator for honeyfiles or file honeybits
```

```
contentgen:
```

~~#~~Neural Network Details

```
rdpconn:
```

```
user: admin
```

```
pass: 12345
```

```
domain: example.com
```

```
template: "screen mode id:i:2\ndesktopwidth:i:1024\ndesktopheight:i:768\nuse
```

```
# server: 192.168.1.66 # Default is 'honeypot addr'
```

~~#~~EMAIL

```
txtemail:
```

```
user: JOHN DOE
```

```
pass: iLoveMachineLearning
```

```
#From: admin<adel@example.com>Subject: Re: passwordchangeDate: April18th ,2017 at 2
```

```
#training_data
train:
path: /tmp/training-data
change date: 21-4-2018
```

```
#testing_data
test:
path: /tmp/testing-data
change date: 21-4-2018
```

```
honeybits:
#FAKE records in config files
#FAKE AWS FILES
awsconf:
enabled: true
#ENABLED TRUE
profile: devsecops
#USER PROFILE
region: us-east-1
#REGION
accesskeyid: AKIAIOSFODNN7EXAMPLE
#ACCESS KEY
secretaccesskey: wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY
#SECRET KEY
```

```
#FAKE host configuration
hostsconf:
enabled: true #ENABLED
# ip: 192.168.1.66 # Default is 'honeypot addr'
name: mysql-srv #name of configuration
```

```
#Fake records in bash_history
ssh:
enabled: true #ENABLED
```

```
# server: 192.168.1.66 # Default is 'honeypot.addr'
port: 2222                                     #LISTENS ON PORT 2222
user: root                                     #ROOT USERNAME
sshpas: true                                  #PASSWORD ENABLED
pas: admin                                    #PASSWORD

#File get FAKE COMMAND
wget:
enabled: true                                #ENABLED
url: http://192.168.1.66:8080/backup.zip       #URL GET
url: http://192.168.1.66:8080/Training-Examples.zip
url: https://Dropbox.org/Back-up.zip
url: http://Dropbox.com/Training-Examples.zip

#File Transfer Protocol FAKE COMMAND
ftp:
enabled: true                                #ENABLED
# server: 192.168.1.66 # Default is 'honeypot.addr'
port: 2121                                    #FTP listens in port 2121
user: admin                                  #USERNAME
pas: admin                                   #PASSWORD

#OS file synching FAKE COMMAND
rsync:
enabled: true
# server: 192.168.1.66 # Default is 'honeypot.addr'
port: 2222                                    # file transfer port listens on local port 2222
user: root                                    # root user
remotepath: /path/to/source                  # REMOTE PATH
localpath: /path/to/destination              # DESTINATION PATH
sshpas: true                                 # PASSWORD ENABLED
pas: 12345                                   # PASSWORD

#FILE TRANSFER FAKE Command
scp:
enabled: true
```

```
# server: 192.168.1.66 # Default is 'honeypot.addr'
port: 2222                # file transfer port listens on local 2
user: root                # root user
remotepath: /path/to/source # REMOTE PATH
localpath:  /path/to/destination # DESTINATION PATH
```

#MYSQL FAKE COMMAND

```
mysql:
enabled: true
# server: 192.168.1.66 # Default is 'honeypot.addr'
port: 3306              # MYSQL port listens on local 3306
user: admin              # username
pass: admin              # password
command: show databases # LIST ALL DATABASES
command: SELECT * from TRAINING-DATA
dbname: machinelearning_data
```

#Amazon Web Services FAKE INFORMATION

```
aws:
enabled: true
profile: devops
region: us-east-2
command: ec2 describe-instances
accesskeyid: AKIAIOSFODNN7EXAMPLE
secretaccesskey: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

#CUSTOM honeybits in bash_history

```
custom:
#Telnet to honeypot application on port 80
- telnet 192.168.1.66 80
#FTP and grab the training examples from admin on honeypot application on po
- ftp ftp://Training-Examples:JOHNDOE@192.168.1.66:80
#train CNN model with valuable Training Examples
- python train-CNN.py /Training-Examples
#test CNN model with valuable Testing Examples
- python test-CNN.py /Testing-Examples
```

```
#start node slave on port 6312
- ./sbin/start-slave.sh 192.168.1.69 6321
#run node slave on port 6312 on testing examples - model-v2000-back-up
- python model-v2000-back-up Testing-Examples
#change password of honeypot app to I USE Tensorflow
- sshpass -p 'IUseTensorflow' ssh -p 6321 JOHNDOE@192.168.1.66
- sshpass -p JOHNDOE@192.168.1.66
- tar -cvf - TrainingImages/Numbers | ssh JOHNDOE@192.168.1.66 '(cd new_imag
```

7.1.4 honeybits.go

```
// Copyright (C) 2017 Adel "0x4D31" Karimi
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is decentralized in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <http://www.gnu.org/licenses/>.

package main

import (
    "fmt"
    "github.com/spf13/viper"
    - "github.com/spf13/viper/remote"
    "io/ioutil"
    "math/rand"
    "os"
    "os/exec"
    "runtime"
```

```
"strings"
"time"
"github.com/0x4D31/honeybits/contentgen"
)

func check(e error) {
    if e != nil {
        os.Stderr.WriteString(fmt.Sprintf("Error: %s\n", e.Error()))
    }
}

func loadCon() (*viper.Viper, error) {
    // Reading config values from environment variables and then getting
    // the remote config (remote Key/Value store such as etcd or Consul)
    // e.g. $ export HBITS_KVSPROVIDER="consul"
    //           $ export HBITS_KVSADDR="127.0.0.1:32775"
    //           $ export HBITS_KVSDIR="/config/hbconf.yaml"
    //           $ export HBITS_KVSKEY="/etc/secrets/mykeyring.gpg"
    conf := viper.New()
    conf.SetEnvPrefix("hbits")
    conf.SetEnvKeyReplacer(strings.NewReplacer(".", "_"))
    conf.SetDefault("kvsprovider", "consul")
    conf.SetDefault("kvsdir", "/config/hbconf.yaml")
    conf.SetDefault("path.bashhistory", "~/.bash_history")
    conf.SetDefault("path.hosts", "/etc/hosts")
    conf.SetDefault("path.awsconf", "~/.aws/config")
    conf.SetDefault("path.awscred", "~/.aws/credentials")

    kvsaddr := conf.GetString("kvsaddr")
    kvsprovider := conf.GetString("kvsprovider")
    kvsdir := conf.GetString("kvsdir")

    // If HBITS_KVSKEY is set, use encryption for the remote Key/Value Store
    if conf.IsSet("kvskey") {
        kvskey := conf.GetString("kvskey")
```

```
conf.AddSecureRemoteProvider(kvsprovider, kvsaddr, kvmdir, kvskey)
} else {
conf.AddRemoteProvider(kvsprovider, kvsaddr, kvmdir)
}
conf.SetConfigType("yaml")
if err := conf.ReadRemoteConfig(); err != nil {

// Reading local config file
fmt.Print("Failed reading remote config. Reading the local config file...\n")
conf.SetConfigName("hbconf")
conf.AddConfigPath("/etc/hbits/")
conf.AddConfigPath(".")
if err := conf.ReadInConfig(); err != nil {
return nil, err
}
fmt.Print("Local configuration file loaded.\n\n")
return conf, nil
}
fmt.Print("Remote configuration file loaded\n\n")
return conf, nil
}

func rndline(l []string) int {
s1 := rand.NewSource(time.Now().UnixNano())
r1 := rand.New(s1)
rl := r1.Intn(len(l))
return rl
}

func contains(s []string, b string) bool {
for _, a := range s {
if a == b {
return true
}
}
return false
```



```
}
```

```
func linefinder(l []string, k string) int {  
    linenum := 0  
    for i := range l {  
        if l[i] == k {  
            linenum = i  
        }  
    }  
    return linenum + 1  
}
```

```
func honeybit_creator(conf *viper.Viper, htype string, hpath string, rnd str
```

```
switch htype {  
case "ssh":  
    sshserver := conf.GetString("honeypot.addr")  
    if p := &sshserver; conf.IsSet("honeybits.ssh.server") {  
        *p = conf.GetString("honeybits.ssh.server")  
    }  
    honeybit := fmt.Sprintf("ssh -p %s %s@%s",  
        conf.GetString("honeybits.ssh.port"),  
        conf.GetString("honeybits.ssh.user"),  
        sshserver)  
    insertbits(htype, hpath, honeybit, rnd)  
case "sshpas":  
    sshserver := conf.GetString("honeypot.addr")  
    if p := &sshserver; conf.IsSet("honeybits.ssh.server") {  
        *p = conf.GetString("honeybits.ssh.server")  
    }  
    honeybit := fmt.Sprintf("sshpas -p '%s' ssh -p %s %s@%s",  
        conf.GetString("honeybits.ssh.pass"),  
        conf.GetString("honeybits.ssh.port"),  
        conf.GetString("honeybits.ssh.user"),  
        sshserver)  
    insertbits(htype, hpath, honeybit, rnd)
```

```
case "wget":
honeybit := fmt.Sprintf("wget %s",
conf.GetString("honeybits.wget.url"))
insertbits(htype, hpath, honeybit, rnd)
case "ftp":
ftpserver := conf.GetString("honeypot.addr")
if p := &ftpserver; conf.IsSet("honeybits.ftp.server") {
*p = conf.GetString("honeybits.ftp.server")
}
honeybit := fmt.Sprintf("ftp ftp://%s:%s@%s:%s",
conf.GetString("honeybits.ftp.user"),
conf.GetString("honeybits.ftp.pass"),
ftpserver,
conf.GetString("honeybits.ftp.port"))
insertbits(htype, hpath, honeybit, rnd)
case "rsync":
rsyncserver := conf.GetString("honeypot.addr")
if p := &rsyncserver; conf.IsSet("honeybits.rsync.server") {
*p = conf.GetString("honeybits.rsync.server")
}
honeybit := fmt.Sprintf("rsync -avz -e 'ssh -p %s' %s@%s:%s %s",
conf.GetString("honeybits.rsync.port"),
conf.GetString("honeybits.rsync.user"),
rsyncserver,
conf.GetString("honeybits.rsync.remotepath"),
conf.GetString("honeybits.rsync.localpath"))
insertbits(htype, hpath, honeybit, rnd)
case "rsyncpass":
honeybit := fmt.Sprintf("rsync -rsh=\"sshpass -p '%s' ssh -l %s -p %s\" %s:%s",
conf.GetString("honeybits.rsync.pass"),
conf.GetString("honeybits.rsync.user"),
conf.GetString("honeybits.rsync.port"),
conf.GetString("honeybits.rsync.server"),
conf.GetString("honeybits.rsync.remotepath"),
conf.GetString("honeybits.rsync.localpath"))
insertbits(htype, hpath, honeybit, rnd)
```

```
case "scp":
    scpserver := conf.GetString("honeypot.addr")
    if p := &scpserver; conf.IsSet("honeybits.scp.server") {
        *p = conf.GetString("honeybits.scp.server")
    }
    honeybit := fmt.Sprintf("scp -P %s %s@%s:%s %s",
        conf.GetString("honeybits.scp.port"),
        conf.GetString("honeybits.scp.user"),
        scpserver,
        conf.GetString("honeybits.scp.remotepath"),
        conf.GetString("honeybits.scp.localpath"))
    insertbits(htype, hpath, honeybit, rnd)
case "mysql":
    mysqlserver := conf.GetString("honeypot.addr")
    if p := &mysqlserver; conf.IsSet("honeybits.mysql.server") {
        *p = conf.GetString("honeybits.mysql.server")
    }
    honeybit := fmt.Sprintf("mysql -h %s -P %s -u %s -p%s -e \"%s\"",
        mysqlserver,
        conf.GetString("honeybits.mysql.port"),
        conf.GetString("honeybits.mysql.user"),
        conf.GetString("honeybits.mysql.pass"),
        conf.GetString("honeybits.mysql.command"))
    insertbits(htype, hpath, honeybit, rnd)
case "mysqldb":
    mysqlserver := conf.GetString("honeypot.addr")
    if p := &mysqlserver; conf.IsSet("honeybits.mysql.server") {
        *p = conf.GetString("honeybits.mysql.server")
    }
    honeybit := fmt.Sprintf("mysql -h %s -u %s -p%s -D %s -e \"%s\"",
        conf.GetString("honeybits.mysql.server"),
        conf.GetString("honeybits.mysql.user"),
        conf.GetString("honeybits.mysql.pass"),
        conf.GetString("honeybits.mysql.dbname"),
        conf.GetString("honeybits.mysql.command"))
    insertbits(htype, hpath, honeybit, rnd)
```

```
case "aws":
honeybit := fmt.Sprintf("export AWS_ACCESS_KEY_ID=%s\nexport AWS_SECRET_ACCESS_KEY=%s",
conf.GetString("honeybits.aws.accesskeyid"),
conf.GetString("honeybits.aws.secretaccesskey"),
conf.GetString("honeybits.aws.command"),
conf.GetString("honeybits.aws.profile"),
conf.GetString("honeybits.aws.region"))
insertbits(htype, hpath, honeybit, rnd)
case "hostsconf":
hostip := conf.GetString("honeypot.addr")
if p := &hostip; conf.IsSet("honeybits.hostsconf.ip") {
*p = conf.GetString("honeybits.hostsconf.ip")
}
honeybit := fmt.Sprintf("%s      %s",
hostip,
conf.GetString("honeybits.hostsconf.name"))
insertbits(htype, hpath, honeybit, rnd)
case "awsconf":
honeybit := fmt.Sprintf("[profile %s]\noutput=json\nregion=%s",
conf.GetString("honeybits.awsconf.profile"),
conf.GetString("honeybits.awsconf.region"))
insertbits(htype, hpath, honeybit, rnd)
case "awscred":
honeybit := fmt.Sprintf("[%s]\naws_access_key_id=%s\naws_secret_access_key=%s",
conf.GetString("honeybits.awsconf.profile"),
conf.GetString("honeybits.awsconf.accesskeyid"),
conf.GetString("honeybits.awsconf.secretaccesskey"))
insertbits(htype, hpath, honeybit, rnd)
//default:
//custom
}
}

func insertbits(ht string, fp string, hb string, rnd string) {
if _, err := os.Stat(fp); os.IsNotExist(err) {
_, err := os.Create(fp)
```

```
check(err)
}
fi, err := ioutil.ReadFile(fp)
check(err)
var lines []string = strings.Split(string(fi), "\n")
var hb_lines []string = strings.Split(string(hb), "\n")
if iscontain := contains(lines, hb_lines[0]); iscontain == false {
if rnd == "true" {
rl := (rndline(lines))
lines = append(lines[:rl], append([]string{hb}, lines[rl:]...)...)
} else if rnd == "false" {
lines = append(lines, hb)
}
output := strings.Join(lines, "\n")
err = ioutil.WriteFile(fp, []byte(output), 0644)
if err != nil {
fmt.Printf("[failed] Can't insert %s honeybit, error: \"%s\"\n", ht, err)
} else {
fmt.Printf("[done] %s honeybit is inserted\n", ht)
}
} else {
fmt.Printf("[failed] %s honeybit already exists\n", ht)
}
}

func honeyfile_creator(conf *viper.Viper, fp string, ft string, template string) {
if _, err := os.Stat(fp); err == nil {
fmt.Printf("[failed] honeyfile already exists at this path: %s\n", fp)
} else {
data := contentgen.Textgen(conf, ft, template)
err := ioutil.WriteFile(fp, []byte(data), 0644)
if err != nil {
fmt.Printf("[failed] Can't create honeyfile, error: \"%s\"\n", err)
} else {
fmt.Printf("[done] honeyfile is created (%s)\n", fp)
}
}
```

```
}  
}
```

```
func honeyfile_monitor(fp string, cf string, m string) {  
    switch m {  
    case "auditd":  
        if runtime.GOOS == "linux" {  
            searchString := fmt.Sprintf("-w %s -p rwa -k honeyfile", fp)  
            out, err := exec.Command("auditctl", "-l").Output()  
            check(err)  
            outString := string(out[:])  
            if strings.Contains(outString, searchString) == false {  
                //pathArg := fmt.Sprintf("path=%s", fp)  
                //err := exec.Command("auditctl", "-a", "exit,always", "-F", pathArg, "-F",  
                err := exec.Command("auditctl", "-w", fp, "-p", "wra", "-k", "honeyfile").Run()  
                check(err)  
                fmt.Printf("[done] auditd rule for %s is added\n", fp)  
            } else {  
                fmt.Print("[failed] auditd rule already exists\n")  
            }  
        } else {  
            fmt.Print("[failed] honeybits auditd monitoring only works on Linux. Use go-  
        }  
  
    case "go-audit":  
        if _, err := os.Stat(cf); err == nil {  
            fi, err := ioutil.ReadFile(cf)  
            check(err)  
            var lines []string = strings.Split(string(fi), "\n")  
            rule := fmt.Sprintf(" -a exit,always -F path=%s -F perm=wra -k honeyfile"  
            if iscontain := contains(lines, rule); iscontain == false {  
                ruleline := linefinder(lines, "rules:")  
                lines = append(lines[:ruleline], append([]string{rule}, lines[ruleline:]...))  
                output := strings.Join(lines, "\n")  
                err = ioutil.WriteFile(cf, []byte(output), 0644)  
                if err != nil {
```

```
fmt.Printf("[failed] Can't add go-audit rule, error: \"%s\\n\", err)
} else {
fmt.Printf("[done] go-audit rule for %s is added\\n", fp)
}
} else {
fmt.Print("[failed] go-audit rule already exists\\n")
}
} else {
check(err)
}
}
}

func main() {

conf, err := loadCon()
check(err)

var (
bhrnd      = conf.GetString("randomline.bashhistory")
cfrnd      = conf.GetString("randomline.conf")
bhpath     = conf.GetString("path.bashhistory")
hostpath   = conf.GetString("path.hosts")
awsconfpath = conf.GetString("path.awsconf")
awscredpath = conf.GetString("path.awscred")
)

// Insert honeybits
// [File]
if conf.GetString("honeyfile.enabled") == "true" {
switch conf.GetString("honeyfile.monitor") {
case "go-audit":
configfile := conf.GetString("honeyfile.goaudit-conf")
if traps := conf.GetStringSlice("honeyfile.traps"); len(traps) != 0 {
for _, t := range traps {
tconf := strings.Split(t, ":")
```

```
honeyfile_creator(conf, tconf[0], tconf[1], tconf[2])
honeyfile_monitor(tconf[0], configfile, "go-audit")
}
}
case "auditd":
if traps := conf.GetStringSlice("honeyfile.traps"); len(traps) != 0 {
for _, t := range traps {
tconf := strings.Split(t, ":")
honeyfile_creator(conf, tconf[0], tconf[1], tconf[2])
honeyfile_monitor(tconf[0], "", "auditd")
}
}
case "none":
if traps := conf.GetStringSlice("honeyfile.traps"); len(traps) != 0 {
for _, t := range traps {
tconf := strings.Split(t, ":")
honeyfile_creator(conf, tconf[0], tconf[1], tconf[2])
}
}
default:
fmt.Print("Error: you must specify one of these options for honeyfile.monito
}
}
// [Bash_history]
///// SSH
if conf.GetString("honeybits.ssh.enabled") == "true" {
if conf.GetString("honeybits.ssh.sshpass") == "true" {
honeybit_creator(conf, "sshpass", bhpath, bhrnd)
} else {
honeybit_creator(conf, "ssh", bhpath, bhrnd)
}
}
}
///// WGET
if conf.GetString("honeybits.wget.enabled") == "true" {
honeybit_creator(conf, "wget", bhpath, bhrnd)
}
}
```



```
///// FTP
if conf.GetString("honeybits.ftp.enabled") == "true" {
honeybit_creator(conf, "ftp", bhpath, bhrnd)
}
///// RSYNC
if conf.GetString("honeybits.rsync.enabled") == "true" {
if conf.GetString("rsync.sshpass") == "true" {
honeybit_creator(conf, "rsyncpass", bhpath, bhrnd)
} else {
honeybit_creator(conf, "rsync", bhpath, bhrnd)
}
}
}
///// SCP
if conf.GetString("honeybits.scp.enabled") == "true" {
honeybit_creator(conf, "scp", bhpath, bhrnd)
}
///// MYSQL
if conf.GetString("honeybits.mysql.enabled") == "true" {
if conf.IsSet("mysql.dbname") {
honeybit_creator(conf, "mysqldb", bhpath, bhrnd)
} else {
honeybit_creator(conf, "mysql", bhpath, bhrnd)
}
}
}
///// AWS
if conf.GetString("honeybits.aws.enabled") == "true" {
honeybit_creator(conf, "aws", bhpath, bhrnd)
}
// [Hosts Conf]
if conf.GetString("honeybits.hostsconf.enabled") == "true" {
honeybit_creator(conf, "hostsconf", hostspath, cfrnd)
}
// [AWS Conf]
if conf.GetString("honeybits.awsconf.enabled") == "true" {
honeybit_creator(conf, "awsconf", awsconffpath, cfrnd)
honeybit_creator(conf, "awscred", awscredpath, cfrnd)
```

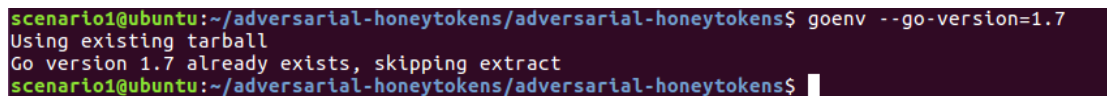
```

}
// Custom bits in bash_history
if cb := conf.GetStringSlice("honeybits.custom"); len(cb) != 0 {
for _, v := range cb {
insertbits("custom", bhpath, v, bhrnd)
}
}
}
}

```

7.2 Scenario1

7.2.1 attachment 1.1 - initializing the Go environment



```

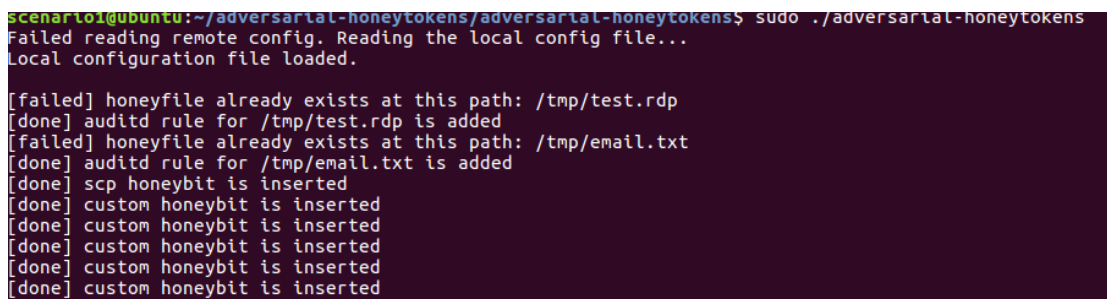
scenario1@ubuntu:~/adversarial-honeytokens/adversarial-honeytokens$ goenv --go-version=1.7
Using existing tarball
Go version 1.7 already exists, skipping extract
scenario1@ubuntu:~/adversarial-honeytokens/adversarial-honeytokens$

```

Figure 7.1: Scenario 1 - attachment 1.1

attachment 1.1 illustrates the initialization of the Go programming virtual environment. Here, we use the `go-env` virtual environment to install the Go compiler *version 1.7* to be used for running this application.

7.2.2 attachment 1.2 - running and deploying the tokens



```

scenario1@ubuntu:~/adversarial-honeytokens/adversarial-honeytokens$ sudo ./adversarial-honeytokens
Failed reading remote config. Reading the local config file...
Local configuration file loaded.

[failed] honeyfile already exists at this path: /tmp/test.rdp
[done] auditd rule for /tmp/test.rdp is added
[failed] honeyfile already exists at this path: /tmp/email.txt
[done] auditd rule for /tmp/email.txt is added
[done] scp honeybit is inserted
[done] custom honeybit is inserted
[done] custom honeybit is inserted
[done] custom honeybit is inserted
[done] custom honeybit is inserted
[done] custom honeybit is inserted
[done] custom honeybit is inserted

```

Figure 7.2: Scenario 1 - attachment 1.2

attachment 1.2 illustrates the running and deployment of the adversarial honeypot tokens, after compilation through the command `go build`. As we can see, the custom tokens have been inserted, as well as the auditd rules.

attachment 1.3 illustrates the deployment location of the email text token inside the `tmp` file system folder.

attachment 1.4 illustrates the insertion of the custom adversarial tokens into the `bash`

7.2.3 attachment 1.3 - deploying email token

```
scenario1@ubuntu:/tmp$ ls
email.txt  systemd-private-2c02699db7b04c91b46b2c2b2f103998-systemd-timesyncd.service-oUmVXR  test.rdp
scenario1@ubuntu:/tmp$
```

Figure 7.3: Scenario1 - attachment 1.3

7.2.4 attachment 1.4

```
scenario1@ubuntu:/home/test$ cat .bash_history
ftp ftp://Training-Examples:JOHNDOE@192.168.1.66:80
sshpass -p 'ILIKEBAYSIAN' ssh -p 6321 JOHNDOE@192.168.1.66
python test-CNN.py /Testing-Examples
python train-CNN.py /Training-Examples
python model-v2000-back-up Testing-Examples
```

Figure 7.4: Scenario 1 - attachment 1.4

history file within */home/test* directory of the file system.

7.2.5 attachment 1.5

```
From: admin <adel@example.com>
Subject: Re: password change
Date: April 18th, 2017 at 21:59:15 GMT+11
To: JOHN DOE (Data Scientist) <JOHN.DOE@example.com>
Cc: security <security@example.com>

Hi,

Ah, sorry I forgot to send you the new address: http://192.168.1.66
I also reset your password (user: JOHN DOE) to the default pass: I-LIKE-BAYSIAN

Please set the MFA (multi-factor authentication) ASAP.

Cheers,
Adel

On April 18th 2017, at 9:57 pm, admin <dave.cohen@example.com> wrote:

Hi admin,

I just wanted to login to the classification system, but I get 404 error. Could you please have a look at it?

Thanks
Dave

The information contained in this email and any attachments is confidential and/or privileged. This email and any attachments are intended to be read only by the person named above. If the
```

Figure 7.5: Scenario1 - attachment 1.5

attachment 1.5 details the email text token documenting the interaction between the system administrator and data scientist, intentionally leaking credentials to entice the adversary.

attachment 1.6 details the email text token documenting the interaction between the system administrator and data scientist, intentionally leaking credentials to entice the adversary.

7.2.6 attachment 1.6

```
scenario1@ubuntu:~$ sudo autrace /home/test
[sudo] password for scenario1:
Waiting to execute: /home/test
Failed to exec /home/test
Cleaning up...
Trace complete. You can locate the records with 'ausearch -i -p 1070'
```

Figure 7.6: Scenario1 - attachment 1.6

7.2.7 attachment 1.7

```
scenario1@ubuntu:~$ sudo autrace /tmp
Waiting to execute: /tmp
Failed to exec /tmp
Cleaning up...
Trace complete. You can locate the records with 'ausearch -i -p 1076'
```

Figure 7.7: Scenario1 - attachment 1.7

attachment 1.7 details the email text token documenting the interaction between the system administrator and data scientist, intentionally leaking credentials to entice the adversary.

7.2.8 attachment 1.8

```
scenario1@ubuntu:~$ sudo ausearch -p 1070 | aureport

Summary Report
=====
[sudo] password for scenario1:
Range of time in logs: 02/12/2018 13:20:41.269 - 02/12/2018 13:20:42.269
Selected time for report: 02/12/2018 13:20:41 - 02/12/2018 13:20:42.269
Number of changes in configuration: 0
Number of changes to accounts, groups, or roles: 0
Number of logins: 0
Number of failed logins: 0
Number of authentications: 0
Number of failed authentications: 0
Number of users: 1
Number of terminals: 1
Number of host names: 0
Number of executables: 1
Number of commands: 1
Number of files: 1
Number of AVC's: 0
Number of MAC events: 0
Number of failed syscalls: 1
Number of anomaly events: 0
Number of responses to anomaly events: 0
Number of crypto events: 0
Number of integrity events: 0
Number of virt events: 0
Number of keys: 0
Number of process IDs: 1
Number of events: 10
```

Figure 7.8: Scenario1 - attachment 1.8

attachment 1.8 details the email text token documenting the interaction between the system administrator and data scientist, intentionally leaking credentials to entice the adversary.

7.2.9 attachment 1.9

```
scenario1@ubuntu:~$ sudo ausearch -p 1076 | aureport

Summary Report
=====
Range of time in logs: 02/12/2018 13:23:14.342 - 02/12/2018 13:23:15.346
Selected time for report: 02/12/2018 13:23:14 - 02/12/2018 13:23:15.346
Number of changes in configuration: 0
Number of changes to accounts, groups, or roles: 0
Number of logins: 0
Number of failed logins: 0
Number of authentications: 0
Number of failed authentications: 0
Number of users: 1
Number of terminals: 1
Number of host names: 0
Number of executables: 1
Number of commands: 1
Number of files: 1
Number of AVC's: 0
Number of MAC events: 0
Number of failed syscalls: 1
Number of anomaly events: 0
Number of responses to anomaly events: 0
Number of crypto events: 0
Number of integrity events: 0
Number of virt events: 0
Number of keys: 0
Number of process IDs: 1
Number of events: 10
```

Figure 7.9: Scenario1 - attachment 1.9

attachment 1.9 details the email text token documenting the interaction between the system administrator and data scientist, intentionally leaking credentials to entice the adversary.

Bibliography

- [1] The Go Programming Language.
- [2] What is a Container, Jan. 2017.
- [3] audit-userspace: Linux audit userspace repository, Jan. 2018. original-date: 2016-02-24T18:31:30Z.
- [4] Store and retrieve encrypted configs from etcd or consul, Feb. 2018. original-date: 2014-10-17T21:08:33Z.
- [5] M. Akiyama, T. Yagi, T. Hariu, and Y. Kadobayashi. HoneyCirculator: distributing credential honeypot for introspection of web-based attack cycle. *Int. J. Inf. Secur.*, pages 1–17, Jan. 2017.
- [6] R. M. Campbell, K. Padayachee, and T. Masombuka. A survey of honeypot research: Trends and opportunities. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 208–212, Dec. 2015.
- [7] N. Carlini and D. Wagner. Defensive Distillation is Not Robust to Adversarial Examples. *arXiv:1607.04311 [cs]*, July 2016. arXiv: 1607.04311.
- [8] N. Carlini and D. Wagner. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. *arXiv:1705.07263 [cs]*, May 2017. arXiv: 1705.07263.
- [9] S. Dowling, M. Schukat, and H. Melvin. A ZigBee honeypot to assess IoT cyberattack behaviour. In *2017 28th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2017.
- [10] A. A. Egupov, S. V. Zareshin, I. M. Yadikin, and D. S. Silnov. Development and implementation of a Honeypot-trap. In *2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIconRus)*, pages 382–385, Feb. 2017.

- [11] S. Francia. viper: Go configuration with fangs, Feb. 2018. original-date: 2014-04-02T14:33:33Z.
- [12] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat]*, Dec. 2014. arXiv: 1412.6572.
- [13] J. D. Guarnizo, A. Tambe, S. S. Bhunia, M. Ochoa, N. O. Tippenhauer, A. Shabtai, and Y. Elovici. SIPHON: Towards Scalable High-Interaction Physical Honeypots. In *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*, CPSS '17, pages 57–68, New York, NY, USA, 2017. ACM.
- [14] honeybits. honeybits, July 2017.
- [15] H. Hosseini, Y. Chen, S. Kannan, B. Zhang, and R. Poovendran. Blocking Transferability of Adversarial Examples in Black-Box Learning Systems. *arXiv:1703.04318 [cs]*, Mar. 2017. arXiv: 1703.04318.
- [16] C. Irvine, D. Formby, S. Litchfield, and R. Beyah. HoneyBot: A Honeypot for Robotic Systems. *Proceedings of the IEEE*, PP(99):1–10, 2017.
- [17] A. Kedrowitsch, D. D. Yao, G. Wang, and K. Cameron. A First Look: Using Linux Containers for Deceptive Honeypots. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '17, pages 15–22, New York, NY, USA, 2017. ACM.
- [18] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv:1607.02533 [cs, stat]*, July 2016. arXiv: 1607.02533.
- [19] M. A. Lihet and V. Dadarlat. How to build a honeypot System in the cloud. In *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, pages 190–194, Sept. 2015.
- [20] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into Transferable Adversarial Examples and Black-box Attacks. *arXiv:1611.02770 [cs]*, Nov. 2016. arXiv: 1611.02770.
- [21] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. *arXiv:1706.06083 [cs, stat]*, June 2017. arXiv: 1706.06083.
- [22] M. Nawrocki, M. Whlisch, T. C. Schmidt, C. Keil, and J. Schnfelder. A Survey on Honeypot Software and Data Analysis. *arXiv:1608.06249 [cs]*, Aug. 2016. arXiv: 1608.06249.

- [23] A. Nguyen, J. Yosinski, and J. Clune. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. *arXiv:1412.1897 [cs]*, Dec. 2014. arXiv: 1412.1897.
- [24] N. Papernot, N. Carlini, I. Goodfellow, R. Feinman, F. Faghri, A. Matyasko, K. Hambardzumyan, Y.-L. Juang, A. Kurakin, R. Sheatsley, A. Garg, and Y.-C. Lin. cleverhans v2.0.0: an adversarial machine learning library. *arXiv:1610.00768 [cs, stat]*, Oct. 2016. arXiv: 1610.00768.
- [25] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv:1605.07277 [cs]*, May 2016. arXiv: 1605.07277.
- [26] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical Black-Box Attacks against Machine Learning. *arXiv:1602.02697 [cs]*, Feb. 2016. arXiv: 1602.02697.
- [27] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. *arXiv:1511.07528 [cs, stat]*, Nov. 2015. arXiv: 1511.07528.
- [28] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597, May 2016.
- [29] S. Rauti and V. Leppnen. A Survey on Fake Entities as a Method to Detect and Monitor Malicious Activity. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 386–390, Mar. 2017.
- [30] M. Ribeiro, K. Grolinger, and M. A. M. Capretz. MLaaS: Machine Learning as a Service. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 896–902, Dec. 2015.
- [31] A. Rozsa, M. Gunther, and T. E. Boult. Towards Robust Deep Neural Networks with BANG. *arXiv:1612.00138 [cs]*, Nov. 2016. arXiv: 1612.00138.
- [32] C. Smith. goenv: Isolated development environments for Go, Feb. 2018. original-date: 2014-07-21T06:30:16Z.

- [33] N. Soule, P. Pal, S. Clark, B. Krisler, and A. Macera. Enabling defensive deception in distributed system environments. In *2016 Resilience Week (RWS)*, pages 73–76, Aug. 2016.
- [34] X. Suo, X. Han, and Y. Gao. Research on the application of honeypot technology in intrusion detection system. In *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*, pages 1030–1032, Sept. 2014.
- [35] F. Tramr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. The Space of Transferable Adversarial Examples. *arXiv:1704.03453 [cs, stat]*, Apr. 2017. arXiv: 1704.03453.
- [36] J. D. Tygar. Adversarial Machine Learning. *IEEE Internet Computing*, 15(5):4–6, Sept. 2011.
- [37] Q. Xiao, K. Li, D. Zhang, and W. Xu. Security Risks in Deep Learning Implementations. *arXiv:1711.11008 [cs]*, Nov. 2017. arXiv: 1711.11008.
- [38] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li. Adversarial Examples: Attacks and Defenses for Deep Learning. *arXiv:1712.07107 [cs, stat]*, Dec. 2017. arXiv: 1712.07107.
- [39] V. Zantedeschi, M.-I. Nicolae, and A. Rawat. Efficient Defenses Against Adversarial Attacks. *arXiv:1707.06728 [cs]*, July 2017. arXiv: 1707.06728.