USING HONEYPOTS IN A DECENTRALIZED FRAMEWORK TO DEFEND AGAINST ADVERSARIAL MACHINE-LEARNING ATTACKS

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

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

Using Honeypots in a Decentralized Framework to Defend Against Adversarial Machine-Learning Attacks

Fadi Younis
Master of Science, Computer Science
Ryerson University, 2018

The market demand for online machine-learning services is increasing, and so to have are the threats to them. Adversarial inputs represent a new threat to Machine-Learning-as-a-Services (MLaaSs). Meticulously crafted malicious inputs can be used to mislead and confuse the learning model, even in cases where the adversary only has access to input and output labels. As a result, there has been increased interest in defence techniques to combat these types of attacks.

In this thesis, we propose a network of high-interaction honeypots as a decentralized defence framework that prevents an adversary from corrupting the learning model, primarily through the use of deception. We accomplish our aim by 1) preventing the attacker from correctly learning the labels and approximating the architecture of the black-box system; 2) luring the attacker away, towards a decoy model, using HoneyTokens; and 3) creating infeasible computational work for the adversary.

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 wonderful colleagues, without whom none of my success would be possible.

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter, we introduce the thesis: adversaries maliciously target classification models (*Section 1.1*). Then, we provide an overview of the problem subject matter (*Section 1.2*), the motivation for solving the problem follows (*Section 1.3*). We then outline the goals of our proposed solution by the end of this thesis (*Section 1.4*). Finally, we end the chapter with an outline for the remainder of this thesis giving a few details about each chapter, (*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, and not confined to one area of industry. With each day, more applications are harnessing the suitability of machine learning in their tract. To meet the ever increasing demand that this field is witnessing. Tech giants, such as *Amazon, Google, Uber, Netflix, Microsoft,* and many others, are providing their adjuncted machine learning products and services in the form of online cloud services, otherwise known as *Machine-Learning-as-a-Service* (MLaaS) [31]. While easily accessible machine learning tools are becoming more popular, the desire to personally customize and build these composite 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, when they simply desire to use them. While some more experienced users still prefer to control how their models are constructed and deployed, companies have undergone the effort to hide the complex internal mechanisms from most of their users, and simply package the services in non-transparent and obfuscated ways. Essentially, they provide their services in the form of a *black-box* [19] [27]. This opaque system container accepts some input and produces an output, but in this system the internally abrasive details of the prediction model are hidden for the user. However, like any deployable application, we cannot assume it is situated in a safe environment, just because the sensitive details are now hidden from all user and potential adversary. There are security flaws and vulnerabilities in every man-made system and machine learning services are no exception. These unseen application details still introduce susceptibility to malicious attack.

1.2 The Problem at a Glance

As stated in the previous section, adversarial machine learning threats exist and are lurking close by. In such an instance, these threats transpire when an attacker misleads and confuses the prediction model inside the cloud computing application, which allow malicious activities to slip-by and go undetected [27]. This drives up the rate of false negatives *FN*, violating model integrity. We call these masqueraded inputs - called *Adversarial Examples* [19], and they represent one of serious threats to cloud services providing MLaaS, such as classifiers. These non-linear inputs look astoundingly similar to the input normally accepted by a linearly designed classifier. However, they only appear this way. They are maliciously crafted to exploit blind spots in the classifier boundary space. Designed to be injected to then exploit, mislead and confuse the learning mechanism acquired by the classifier post-training. They can then compromise the classification model integrity, during the testing phase, when the model handles unseen input. Most known defenses aim at strengthening the classifier's discriminator function by training it on adversarial malicious input ahead of time to make it resistant to these type of inputs. Current proposed defense methods such as *Regularization and Adversarial Training* have proven unsuccessful and ineffective [38], and they cannot be relied upon since they do not generalize well on newer and more potent adversarial inputs. This is particularly true in the case of a black-box setting, where the adversary is limited in knowledge to only input and output labels. Hence, in this thesis we propose a different approach. Our aim will be to develop an adversarial defense framework that poses as a secondary-level of prevention to curb adversarial examples from corrupting the classifier, and it will be used to deceive the attacker.

1.3 Motivation

The market demand for online machine learning services 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 [24]. These masqueraded inputs (Adversarial Examples) represent a recent threat to cloud services providing machine-learning-as-a-service. They are maliciously crafted to exploit *blind spots* in the classifier boundary space. Exploiting these blind sports can be used to mislead and confuse the learning mechanism in the classifier, post model training, for purposes of violating model integrity. As a result, there has been an increased interest in defense techniques to combat them and fortify classifiers against attacks.

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

As stated earlier, some of the previous research on adversarial defense methods has aimed at strengthening the classifier's discriminator function by training it on malicious input beforehand to make it robust. Defense methods, such as *Regularization and Adversarial Training* have had their limitations, as we will show later [33]. This is in particular the case for a *black-box* (*blind-model*) setting, where the adversary has only access to input and output labels. We believe it is necessary to develop an adversarial defense framework to act as a fore-fronting secondary-level of protection to prevent adversarial examples from fooling and evading the classifier, by deceiving the attacker and mitigate the risk of attack. The majority of our efforts are focused on designing a decentralized network of *High-Interaction Honeypots* (HIHP), as an open target for adversaries, acting as a type of *perimeter* defense. This decentralized network of honeypot nodes act as self-contained *sandboxes*, to contain the decoy neural network, collect valuable data, and potentially gain insight into adversarial attacks. We believe this can also confound and deter adversaries from attacking the target model to begin with. Other adversarial defenses can also benefit by utilizing this framework as an additive layer of security to their techniques to protect production servers where learning models reside. Unlike other defense models proposed in literature, we have designed our defense framework to deceive the adversary in three consecutive steps, occurring in strategic order. The information collected from the attacker's interaction with the decoy model could then potentially be used to learn from the attacker, re-train and fortify the deep learning model in future training iterations, but for now this falls out outside on scope. Our defense approach is motivated by trying to answer the following question:

"Is there a computationally feasible and practical way, within a black-box setting, to fool

an adversary and prevent him/her from interacting and learning the behavior of a prediction model before an actual 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 and Contributions

Our thesis is as follows:

"Given a deep learning classification model T, within a private black-box setting, and a intrusive attacker with an adversarial input $\vec{x}^{*}$*, capable of violating model integrity and misclassifying target label y; there exists a defense framework* $D_{defenses}$ *to support existing defenses systems, to 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 model's T behavior and then corrupting it."*

The purpose of this work is to investigative the thesis above and work towards an appropriate defense framework that implements it. The following are our objectives and contributions:

- Propose and work towards an adversarial defense approach that will act as a secondary-level of protecting to *cloak* and reinforce existing adversarial defense mechanisms. It aims to: 1) prevent an attacker from correctly learning the classifier labels and approximating the correct architecture of the black-box system; 2) lure attackers away from the target model towards a decoy model, and re-channel *adversarial transferability*; 3) create infeasible computational work for the adversary, with no functional use or benefit, other than to waste his resources and distract him while learning his techniques.
- Provide an architecture and extend implementation of the *Adversarial Honey-Tokens*, their designs, features, usage, deployment benefits, and evaluations.

1.5 Overview

This thesis has 6 chapters. It is divided as follows:

- *Chapter 1 - gives a brief introduction to the problem at hand and its contextual setting. This chapter also gives an overview of the thesis goals, contributions, and a breakdown of the thesis 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 authored by other researchers on the topic of adversarial black box defenses, as well as specific work done with honeypot technologies.*
- *Chapter 4 - outlines the design and architecture of the decentralized defense approach. It also gives insight into the approach's assumptions, setup,design decisions, environment and limitations. It gives a break down of the how adversary launches an attack, and how the frameworks fares in defending against it.*
- *Chapter 5 - provides an extended implementation of the Adversarial Honey-Tokens, its features, usage, deployment and benefits.*
- *Chapter 6 - summarizes the thesis, gives an overview of the contribution made, and suggests future research directions.*

Chapter 2

Background

The aim of this chapter is to introduce the main problems and challenges involved in defending against adversarial examples. We formulate and explain important concepts so that they can be used to understand and navigate the upcoming chapters. First, we discuss deep neural networks, as well as their security, architecture, and 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, processes of generation, impact and known defenses against them *(Section 2.2)*. Next in (*Section 2.3*), we discuss adversarial transferability, the 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 background section on Honeypots, their classification, deployment modes, roles and responsibilities, advantages and disadvantages, levels of interaction, uses, and notably how honeypots fit into our defense solution (*Section 2.4*).

2.1 Security of Deep Learning Network

We cannot start discussing attacks on deep neural networks without first exploring what a *Deep Neural Network (DNN)* actually is, and how it behaves. Moving forward, we will thoroughly discuss to the threats that jeopardize the integrity of deep neural networks.

2.1.1 Deep Neural Networks

According to [27], 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 tensor, a stream of text, video, etc. These structures (DNNs) are used to allow 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* hidden learning 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 that they contain only one or two hidden layers. Having a few layers limits the ability of the model to learn and adapt to intricate features and solve complex problems. Amongst the countless uses for DNNs' is their utility in building image classification systems that can identify an object from the its intricate edges, features, depth and colors. All of that information is processed in the hidden layers of the model, known as the *deep* layers. As the number of these *deep* layers increases, so does the capability of the DNN to model and solve complex tasks. For a detailed illustration of a typical DNN architecture, see Figure 2.1 from [27]. Simply expressed, a DNN is composed of a series of parametric functions. Each parametric function f_i represents a hidden layer i in the DNN, where each layer i compromises a sequence of *perceptrons* (artificial neurons), which a processing units that can be modeled into chain sequence of computation. Each neuron maps an input x to an output y, $f : x \longrightarrow y$, using an *activation function f(φ)*. With each layer, every neuron is influenced by a parameterized *weight vector* represented by θ_{ij}. The weight vectors holds the knowledge of the DNN when it comes to training and preparing the model F. A DNN computes and defines model a F for an input $\vec{x}$ as follows [27]:

$$F(\vec{x}) = f_n(\theta_{ij}, f_{n-1}(\theta_{n-1,j}, \cdots, f_2(\theta_{2j}, f_1(\theta_{1j}, \vec{x}))))$$

The *training phase* in the DNN occurs when model F learns the values for the hidden layer neuron weights $\theta_F = \{\theta_{1j}, \theta_{2j}, ..., \theta_{nj}\}$. The model is given a large of set of input and output pair examples represented by $(x^{'}, y^{'})$, and works by adjusting the weights and reducing (minimizing) the difference (error) in the cost function C[f] between the predicted value of $F(x^{(i)})$ example and what its true labeled output $F(y^{(i)})$ should be (usually done using back propagation) [27]. The *testing phase* occurs when DNN model F is done training and is deployed with fixed parameters θ_{ij} to make generalized predictions on inputs not seen during training. The weighted parameters represent DNN knowledge progressively acquired in the *training phase*. What usually happens is that the DNN would generalize on this unseen input and make accurate predictions for the output, using only what it's learned during training [27].

There are several types of deep neural network architectures that define the network layers, which vary in functionality and dexterity, depending on the problems they are called upon 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 Models* (another type of RNN) which models sequential information in a Time Series sequence; 4) *Auto-Encoders*; 5) *Re-enforcement Learning*; 6) *Generative Artificial Networks* (GAN). We do not spend time going into depth explaining each type as it falls out of scope. However, our thesis relies heavily on using a DNN model (as a decoy), which we will explore all this further in *Chapter 4*.

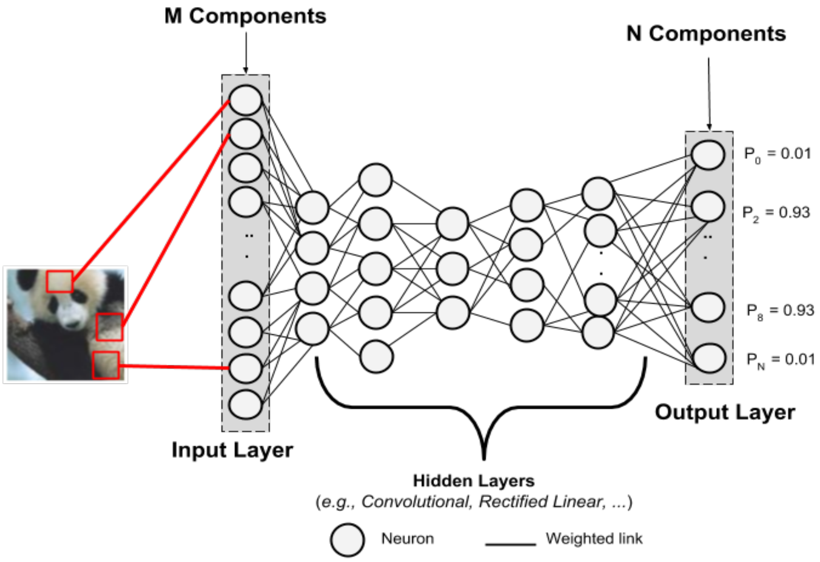


Figure 2.1: A typical DNN architecture. In this particular model, the DNN recognizes images of handwritten digit integers 1...9 and calculates the probability of that image being in one of the N=10 classes [27].

2.1.2 Adversarial Deep Learning

In recent deep learning literature, there has been a lot of works that has focused on deploying deep neural networks in malicious environments, in which the network is potentially exposed to numerous attacks [13] [19] [38]. At the center of these threats are *Adversarial Examples. Please note, because we dedicate an entire later section to depicting what these examples are, we will focus only on the essential idea behind them for now.* 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 the model is completed [29]. They are injected in order to circumvent the learning mechanism acquired by the DNN with the goal of misclassifying a target label. They are crafted with carefully articulated perturbations, added to the input $\vec{x}+\delta\vec{x}$, that *forces* the DNN to display a different behavior than intended, chosen by the adversary [29]. It is important to note that the magnitude of perturbations must be kept *small enough* to have a significant effect on the DNN, yet remain unnoticed by a human being. These adversarial exploitations vary in their motivation for corrupting a DNN classifier, however some of the most common incentives range from simply reducing the confidence of a target label to a arbitrary source-label misclassification [29]. Confidence reduction entails reducing the accuracy on a label y for a particular input x in a the testing pair $(x^{'}, y^{'})$. By contrast, source label misclassification involves having the model classify an input x as a chosen target label y_{target}, different from the original (and intended) true source label y_{true}.

For any attack to be successful, it requires the adversary to have previous knowledge of the DNN architecture, preferably a strong one. This knowledge can perfect white-box attacks, partial black-box attacks or no-knowledge blind. However, it is possible to attack a DNN model F with limited knowledge in hand. In past work, such as [29], the attacker was able to approximate the architecture of a target model, F_{target}, in a black-box setting, and create a substitute training model, which was then used to craft adversarial examples that generalize on both models. These example were transferred back to target model, by way of *adversarial transferability* [29] - a very powerful property, which enables an adversary to transfer malicious examples between models to evade a target classifier model (please see Section 2.3 for more details).

2.1.3 Deep Learning Threats

While deep learning networks have gathered much attention in terms of capability to solve complex 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 kind:

- *Modified Training Data* - commonly known as a *causative* or *poisoning* attack, in which the adversary influences or manipulates the training data set χ, with a transformation. This modification could entail control over a small portion or an important determinant feature dimension D_i in the training data. With this type of attack advance, the attacker can mislead the learner in order to produce a badly classifier, which the adversary exploits post training [16].

- *Poorly Trained DNN Models* - although considered an oversight, rather than blamed on an external adversary. A perfunctory trained DNN could be due to several reasons. Most of the time, developers credulously use DNNs prepared and trained by others.

These same DNNs could have hidden vulnerabilities ripe for exploitation, which can become easy targets for manipulation by adversaries during deployment [37].

- *Perturbed Input Image* - commonly known as *adversarial examples* [19], attackers are also known to attack DNN models, during testing, by constructing malformed input to evade the learning mechanism learned of the DNN classifier. This is known as an *evasion attack* [16].

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 defense techniques against adversarial examples from the literature.

2.2.1 Adversarial Example Definitions

As mentioned, machine learning models are vulnerable to adversarial attacks that seek to destabilize the neural network's ability to generalize new input; which jeopardizes the security of the model. From what we learned from the authors in [16], these attacks can either occur during the training phase 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 a poorly trained model, which cannot be easily detected. These disruptive anomalies are known as *adversarial examples*. See Figure 2.2 below for an illustration of adversarial examples

Adversarial examples are slightly perturbed versions of regular input samples normally accepted classifiers. They are maliciously designed to have the same appearance as regular input, from a human's point of view, at least. These masquerading 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 are widely exploitable, if left unattended. To better understand them, consider this example: given an input sample $\vec{x}$ classified with function C, such that $C(\vec{x}) = \ell$, producing output ℓ, that was correctly classified by model $A(\cdot)$, we say the perturbed input sample $\vec{x^*}$, so that $C(\vec{x^*}) = \ell$, we say x'

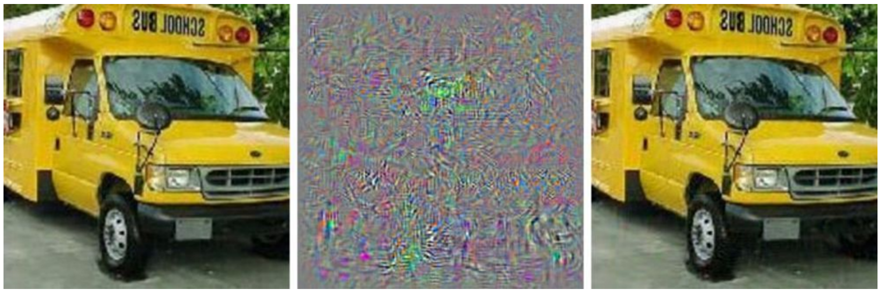


Figure 2.2: Adversarial Examples - Input $\vec{x}$ (left), modification $\delta+\vec{x}$ controlled by ε (middle) which controls the magnitude of modification in the image, generating the adversarial evading sample $\vec{x}^*$(right). As you can see, both bus images look astoundingly similar.

is an adversarial example of x such that $A(x^{'}) = A(x)$. For an illustration of what adversarial examples are, see Figure 2.3 below for a better understanding of what is construed by an adversarial example [26]. Classification models are considered *robust* if their classification ability is unaffected by the presence and exploits of adversarial examples.

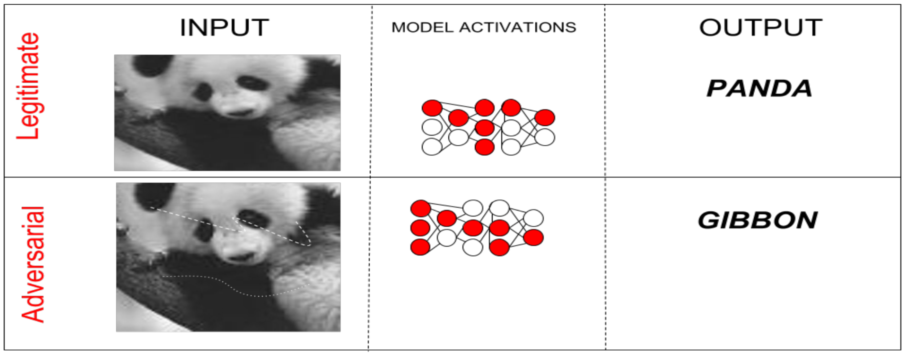


Figure 2.3: An Adversarial Example is generated by modifying a legitimate input sample $\vec{x}$, in such a way which would cause to the classifier to mislabel it as z instead of y, where as a human wouldn't notice a difference.

2.2.2 Adversarial Example Properties

In the beginning of the previous section, we mentioned that adversarial examples $\vec{x}^*$ possess an appearance similar or *close* to the original input samples $\vec{x}$. Normally used, although not the only form of measurement. This measure of *closeness* or *similarity* between the pair of original and modified input is known as the p-norm distance $\| x \|_p$. This degree of closeness could be l_2, which is the *Euclidean Distance* between two pixels in an input sample x, l_∞, which is the absolute or max change made to a pixel in x, or l_1; which is the total number of pixel changes made to the input sample x [9]. If the measure of distortion in any of the previous metrics of closeness is small, then those input samples must be visually similar to each other, which made them a prime candidate for adversarial example generation. 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 $F(\cdot)$ will misclassify on model $G(\cdot)$, even if they are trained with two different neural net architectures, or dissimilar training training sets [9]. This intriguing property of transferability enables the target and original model to both generalize on the same adversarial example and allow both models $F(\cdot)$ and $G(\cdot)$ to assign the adversarial example to the same class. This is because the perturbation examples has found the perfect balance of distortion to align itself with both models. In order to understand how this property of transferability is possible, we must first understand why it occurs and where these adversarial examples originate from.

Work in [13] shows seems to shed some light on their origins. The authors found that adversarial examples are not scattered out randomly, but are systematic in nature and 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 [13] argue that as the higher dimensionality space of any 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 [36] argue that due to the shared dimensionality property, the decision boundary of any two models that have an adversarial example between them, will transfer from model $F(\cdot)$ and $G(\cdot)$, and must be 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, which is simply why they even exist. Many papers have offered hypotheses on their origin. The most popular propositions includes work in [13], which states that some models trained are too *linear* for a n-dimensional data set χ with very little non-linearity. Non-linearity here refers to the discontinuous mappings between output, y, and input, x, where one input can possibly have multiple solutions, which the model is the unprepared for. This could explain why adversarial examples are common in small regions within the de-

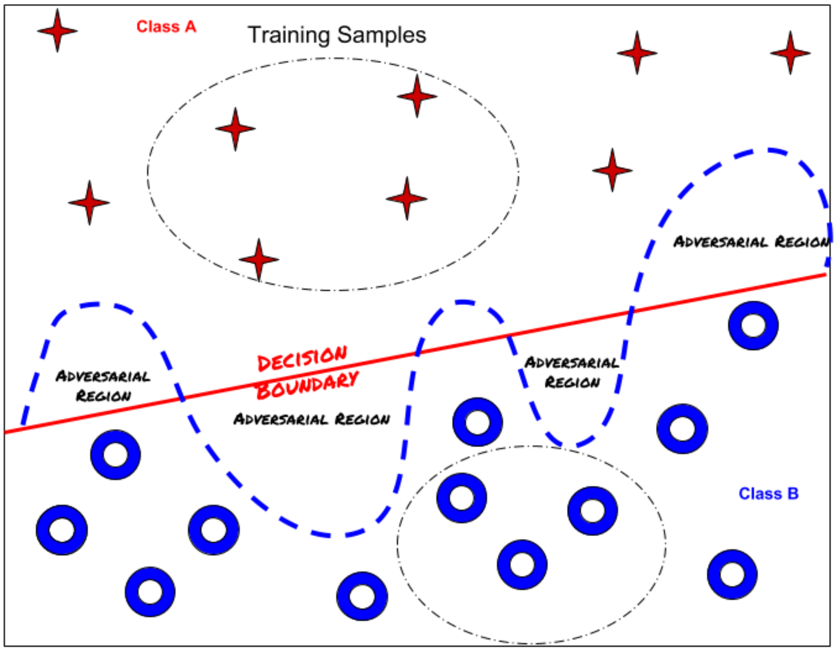


Figure 2.4: Adversarial Regions in Classifier Decision Space - adversarial examples are not scattered out randomly, but are systematic in nature and occur in large and continuous regions in the decision space.

cision space. Opposed to the latter, work in [26], explains that adversarial examples could be the result of deep learning models not being “flexible” for certain tasks and inputs. *Tramér et al.* in [36] suggested that perhaps adversarial examples exist simply due the p-norm distance in the model’s separating decision boundary being in some cases longer than the distance between two models F and G 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 [19] [27]. In order for us to understand how these minatory objects are generated, we must first understand the individual components needed in their creation. Consider the following notation [19]:

$\boldsymbol{x}$ - a clean input (untampered) from some testing dataset D_{test}, typically a 3-D tensor (width

× height × depth). Generally, the input sample image pixel values range between *0* and *255*.

y(true) - the corresponding true label for input ***x***.

J(x, 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, in the direction of its gradient.

ε***, n*** - the hyper-parameters added to influence the model F to create the adversarial example $\vec{x^*}$. 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 human observer.

As mentioned, there are several methods to generate adversarial examples. We won't expand and explain each one, however we will focus most of our efforts exploring the *Jacobian-based Salience Map Approach* (JSMA) [28], which the adversary in our thesis uses as the main adversarial training algorithm to generated the adversarial examples in his attack, after the construction of the substitute model. The following are some methods to generate adversarial examples:

Fast Gradient Sign Method (FGSM) - introduced by Goodfellow et al., in [13], 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( )*** used to train the prediction model, by taking the gradient of the model *F*, with respect to every feature $x_{m,n}$ found in the neighborhood of training input sample $\vec{x}$, which the adversary wants to force a label misclassification of. The best adversarial example or perturbed image $\vec{x^*}$ is computed from the input $\vec{x}$ by solving the following optimization problem [25]:

$$\vec{x^*} = \vec{x} + \varepsilon \cdot sign(\bigtriangledown_{\vec{x}J\ (x,\ y(true))})$$

Here, the ε represents the factor responsible for calibrating magnitude of perturbation in the input sample $\vec{x}$, while *J (x, y(true))* represents the cost function we wish to maximize loss for input sample $\vec{x}$, while keeping ε small. The value of ε needs to be optimized, since a large value of ε to compute the adversarial example will cause the sample to be misclassified by F, 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 $\vec{x^*}$ and can be completed in 1 iteration step.

Basic Iterative Method (BIM) - considered to be an extension of the FGSM method above. Contrastingly, 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 most optimized adversarial example is computed in its final form by solving the following optimization problem [25]:

$$\vec{x^*_{n+1}} = \vec{x_n} + \varepsilon sign(\bigtriangledown_{\vec{x}J\ (x,\ y(true))})$$

Jacobian-based Salience Map Approach (JSMA) - introduced first by Papernot at al. in [28]. This non-iterative (but slower) and computationally intensive generation approach works by perturbing the feature of an input sample $\vec{x}$ that has large *adversarial saliency scores*, this method is used in our targeted attack. Basically, the saliency score represents the goal of taking a sample $\vec{x}$ with a important or noticeable features from its source class *f($\vec{x}$)* = $y_1(true)$ across the decision boundary to a target class *f($\vec{x}$)* = $y_2(true)$.

First, the adversary computes the *Jacobian Matrix* $J(f)(\vec{x}) \in R^{(m \times n)}$, which are all the first-order derivatives $f'(\vec{x})$ 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 in $\vec{x}$. To compute our saliency map, the adversary computes the following for each input feature matrix item i, given to us by, see equation below [25].

$$S(\vec{x}, t)[i] = \begin{cases} 0 \text{ if } \quad \frac{\partial f_i(\vec{x})}{\partial \vec{x_i}} < 0 \quad \text{or} \quad \sum_{j \neq t} \frac{\partial f_i(\vec{x})}{\partial \vec{x_i}} > 0 \\ (\frac{\partial f_i(\vec{x})}{\partial \vec{x_i}}) \quad |\sum_{j \neq t} \frac{\partial f_i(\vec{x})}{\partial \vec{x_i}}| \quad \text{otherwise} \end{cases} \tag{2.1}$$

The value t is the target class $y(target)$ that we 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 in several iterations until we cause the misclassification to occur on the target class *y(true)*, or we reach the maximum number of perturbed features.

There also other adversarial examples generation methods, such as the *iterative-less-likely-class-method* [19]. 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 in our thesis.

2.2.5 The Adversarial Optimization Problem

As we saw in the previous section, 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 computational cost involved. In the general case, adversarial examples are generated by solving a hard optimization problem similar to the one below [27]:

$$\vec{x^*} = \vec{x} + argmin\{\vec{z} : \hat{O}(\vec{x} + \vec{z}) \neq \hat{O}(\vec{x})\} = \vec{x} + \delta_x$$

Where $\vec{x}+\delta_{\vec{x}}$ represents the least possible amount of *noise* added to cause a perturbation, 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_{target}, with a queried adversarial sample, 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 and drop in target label accuracy the attacker is after is achieved by adding the least amount of possible noise $\delta_{\vec{x}}$ to the input $\vec{x}$, in order to be unnoticeable by humans, but just enough to mislead the DNN. Solving for $\vec{x^*}$ is an optimization problem that is not easy to solve since it is *non-linear*, where multiple true solutions exist, and *non-convex*, where there not so easy to find. 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. In convex-type problems, 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 constraints. Here, optimality has become a problem, and an exponential amount of time and variables are required to find a feasible solution, where many indeed exist. Figure 2.5 illustrates *convex* vs. *non-convex* optimization. By preventing the attacker from learning anything about the model T_{target} 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 T_{decoy}, which has no real value. Generating these adversarial examples is already 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 a highly infeasible task done in vanity.

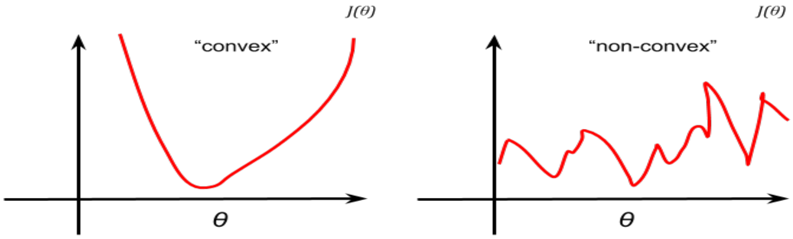


Figure 2.5: Convex vs. non-convex optimization, where one solution exists as the *global minimum*(left), and multiple solution exist *local minimums*(right)

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. 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 $X_{m,n} = (x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{1,n})$, embedded within the image that the DNN model recognizes and learns, which it then assigns to its correct class label. However, according to [24] it was shown that the DNN models can be *tricked* and convinced that a *slightly* perturbed image or input that should otherwise be unrecognizable and consequently rejected by the neural network, can be *forced* to be generalized and accepted as a recognizable member of a class in the targeted model. 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 from any user. This creates calamity, because any chosen input unrecognizable to the model can be *transformed* and classified with high confidence causing a *(false positive)*, and an input recognizable to the model can be classified with low confidence *(false negative)*, violating the integrity of a prediction model, eventually making it unusable.

For instance, 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 [19]. 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 [19].

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 augmenting training data set examples D with added adversarial examples, better known as *adversarial training.* The latter essentially works as follows: the model F is trained on both clean x and perturbed samples $\acute{x}$. The purpose of this method is to increase the robustness of the training model in all gradient directions, which means it should be able to classify an input sample to its true class label y_{true}, and detect any perturbations $\vec{x^*}$ it may encounter. Below, we briefly explain some of the other proposed methods:

- **Network Distillation:** also know as *defensive distillation* [29] [38], originally designed as defense method to reduce the size of a DNN by transferring knowledge from a large DNN to a smaller one, to improve robustness. The authors in [29] found that the adversary usually targets the model's surface sensitivity to perturbations, which is what their solution attempted to rectify and block.

- **Adversarial Re-training:** proposed by Papernot, considered to be another way to make deep neural networks more robust, by regularizing the neural network [13], as well as improving the precision.

- **Input Reconstruction:** adversarial examples can be reclaimed and transformed to clean data, and make them harmless to the deep neural network. The authors in [38] used a *de-noiser* that detects adversarial examples and removes the perturbation from the input and converts it to its original form.

- **Classifier Robustifying:** a robust DNN 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 with *Radial Basis Function* (RBF) kernel $K(x_i, x_j) = exp(-\gamma\|x_i - x_j\|)^2, \gamma > 0$ to improve the distribution of data samples and robustify the model. This is done by *filtering* the input features that similar and dissimilar to the features space against a constraint, removing with that unwanted 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 in pre-training. The authors in [38] suggested using *DeepSafe*, which uses *RELU-plex* in their defensee. 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 using adversarial examples on classification models. Let us first introduce the concept of Adversarial Transferability.

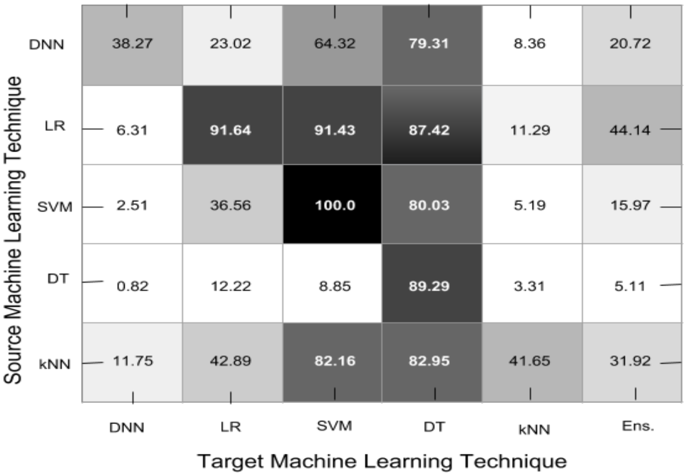
2.3.1 Adversarial Transferability

According to the authors in [36], 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 $\vec{x}$, the adversarial examples $\vec{x^*}$ generated to confuse and mislead one model m can be *transfered* and used to confuse other models $n_1, n_2, n_3, ..., n_i$, that are of homogeneous or even heterogeneous classifier architectures. This mysterious phenomena is mainly due to the determining 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]. 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 [13], 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 example set D, the more likely that the subspaces will intersect significantly, guaranteeing the transfer of samples between the two subspaces [13].

According to the above hypothesis, transferability holds true between two models *as long as both models share a similar purpose or task* [26]. Knowing this, an attacker can leverage the property of transferability to launch an preemptive attack, by training a local substitute classifier model F on sample testing data pairs $(x^{'}, y^{'})$, that the chosen remote target classifier T_{target} were generalized on. Collecting these testing pairs can be formed into a training dataset $D_{training}$ of size N of similar dimensions and content. With the latter we can produce adversarial examples $\vec{x^*}$. It is also worth noting that the success rate of transferability



Source Machine Learning Technique	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	15.97
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 2.6: 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 [26].

varies depending on the type of remote target classifier the examples $\vec{x^*}$ are being transferred to. The Figure 2.6 above [26] illustrates the transferability matrix for adversarial examples generated with a classifier of various types of prediction techniques i and transfered to a target classifier trained with technique j. As it can be seen in Figure 2.6 (darkened box) 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 n also effect model m. 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 [27], which is the same attack we also used for our designed adversary. 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 [36], as we'll see in the coming sections.

Adversarial Examples cannot always be transferred, but to measure the success of transferability of adversarial examples between m and other models n_i, 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 transfered from the substitute model back to the original model [27]. The success rate refers to the portion of perturbations that will be misclassified by the substitute training model F, while the transferability rate refers to those same perturbation samples that will misclassified and generalized by the target model, when transferred form the substitute model after training is done.

2.3.2 Black-Box Threat Model

To explain a black-box threat model, we start by the term *black-box* system concept. 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 remaining invisible. The functionality of the black-box can only be understood by observation, which is what the attacker depends on to begin his attack. Figure below 2.7 illustrates a basic black-box system.

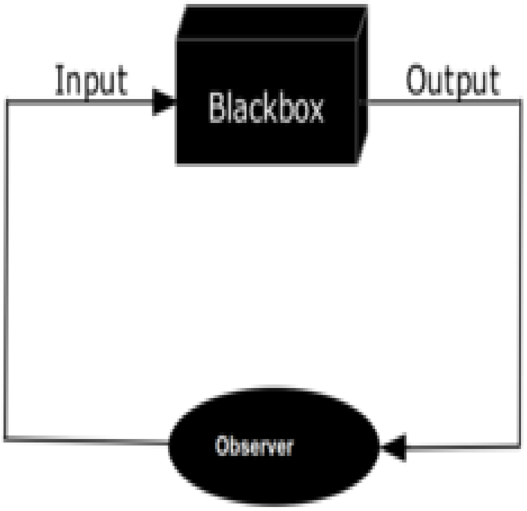


Figure 2.7: A Simple black-box System - construed in terms of inputs x and outputs y, with the internal mechanisms of the system $f(x) = y$ transforming x into y remains invisible

The black-box threat model is by extension a black-box system. In our thesis, we are attempting to prevent the attacker from polluting the target classifier T_{target}, by blocking transferability and access to the target model to change the prediction on the class label y. 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 [27].

Generally, in order to attack the model, in a 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_{target}, 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 substitute learning model and queries the target classifier (represented by the 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. Querying the *Oracle O* for input $\vec{x}$, which represents the only capability available to the attacker, as in the black-box model no access to the Oracle internal details is possible [27]. 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) [27]. What the adversary attempts to do is solve the following optimization problem to generate the adversarial samples, as seen below:

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

The adversary must able to solve this optimization problem by adding a perturbation at an appropriate rate with $\delta_{\vec{x}}$, to avoid human detection. The magnitude ε of the rate must be generated in such a way with the least perturbation possible in $\delta_{\vec{x}}$ to influence the classifier, as well remain undetected by a human [27]. 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 2.2.5*). Further more, removing knowledge of the architecture and training data makes it difficult to find a perturbation that satisfied the condition for successful adversarial examples secretion, where $O\{\vec{x} + \delta_{\vec{x}}\} = O\{\vec{x}\}$ [27].

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 knows in order to 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 but still interesting to compare against.

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 an 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* [29] 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 [10].

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 training model F with synthetic labels S_0 collected by observing the labeling of test samples by 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 can then craft adversarial samples that will be misclassified by the substitute model F [29]. 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 $F(S_0)$ and T_{target}. adversarial examples misclassified by F will be misclassified by the target as well. In our thesis, we find a way to re-channel adversarial transferability and prevent an attack. We plan to accomplish the latter via *deception*.

2.3.5 Transferability of Adversarial Examples in Black-Box Attacks

It was Papernot in [27] [26], 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 knowledge. 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 transferable examples, then transfer them to the unprepared 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.). Work, namely in [9] [21] [27], all reached the same conclusion - adversarial examples will transfer across different models trained on different dataset implementations, with different machine learning techniques.

2.3.6 Black-Box Attack Approach

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 an initial substitute model training set S_0. The adversarial goal of finding a minimal perturbation to misclassify a targeted classifier model T_{target} 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 [27]. What the adversary can do is attempt to mimic or approximate the target *Oracle O* model's architecture, but this requires internal knowledge of the classifier structure, which is not possible considering the inaccessibility under a black-box model scenario [27]. Another benefit to this approach is the fact that most machine learning models require large and expensive training datasets D. This makes incredibly difficult for the attacker to attack a system in such a environment [27], but there are ways around this. The black-box attack strategy consists of the following steps:

Substitute Model Training:

Here, the attacker queries the *Oracle* O with testing sample example input $x^{'}$ and observes the predicted output $y^{'}$. 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 [27]. 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 [27]. The authors in [27] 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 fewer query for labels [27].

Substitute Architecture:

According to the authors in [27], this step is considered crucial in constructing the model, because the author must experiment with different model architectures until he finds the correct one. This is due to the notion that without knowledge of the target model T_{target} architecture, the attacker knows very little about how the system learns and 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 F for the substitute training model $F(S_0)$ is to simply explore and try different variations of the substitute architecture, this continues until we find one that causes a misclassification on the target label [27].

Generating Synthetic Dataset:

Considered as another complication in the path of constructing a successful black-box attack is crafting the dataset used to train the substitute training model F. We could potentially request an infinite number of queries to get the oracle's output $O(x) = y$ for an input sample x [27]. However, this method, although effective in the sense that it creates a suitable training set, it is actually not methodical. This is due to the explicit exposure by interaction and querying *Oracle* O. Creating a large number of queries attracts attention from the defender and makes the adversarial attempts to query suspicious [27].

Substitute Model Training:

Here, we describe the five step algorithm procedure described in [27]. The algorithm is outlined in full in the algorithm above. For a better understanding of how this algorithm is

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

```
Input: $\hat{O}, max_{\rho-1}, S_\rho, \lambda$
Define architecture F
for $\rho \in 0...max_{\rho-1}$ do
    //  Label the substitute training set
    $D \leftarrow \{(\vec{x}, \hat{O}(\vec{x})) : \vec{x} \in S_\rho\}$
    //  Train F on D to evaluate parameter $\theta_F$
    $\theta_F \leftarrow train(F, D)$
    //  Perform Jacobian-based dataset augmentation
    $S_{\rho+1} \leftarrow \{\vec{x} + \lambda \cdot sign(J_F[\hat{O(\vec{x})}]) : \vec{x} \in S_\rho\} \cup S_\rho$
end for
return $\theta_F$
```

used to train substitute model, see below for a brief description of the steps involved [27]:

Initial Collection (step 1): the adversary collects a small set of test samples example (x', y') pairs that will later used to create the initial training set S_0. The test samples are taken by querying *Oracle O* and observing its behavior. These inputs resemble training domain of the target model, in the sense that are taken from the same statistical distribution.

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

Substitute Training (super step) the adversary selects more appropriate substitute models Fp until he finds the most appropriate one, by repeating the following steps:

Labeling (step 3): the labeled samples are collected by querying the *Oracle O* in step 1. These labels are then used to compose the first of several substitute model training set Sp to train the substitute model F in $F(S_0)$.

Training (step 4): the adversary gradually trains the substitute architecture training model $F(S_p)$ selected in step 2 above using known supervised adversarial training techniques with using labeled data Sp collected from step 3.

Augmentation (step 5): the authors augmentation technique in [27] is used on the updated training set Sp 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 increases the substitute model F accuracy reaches satisfaction and mimics 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 [27], 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 black-box oriented adversarial attacks in [15] include: 1) Preprocessing Methods; 2) Regularization and Adversarial Training; 3) Distillation Methods; 4) Classification with Rejection. Let us review a few of these methods:

Preprocessing Methods:

The authors in [29] mention this method as way filter out input $\vec{x}$ determined to be adversarial. The authors argue that input samples images have natural properties, such as high correlations 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 a pretext to filter out malicious perturbations $\delta\vec{x}$. 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's 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 [13] 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 ineffective for both black-box and Blind Threat Models against adversarial examples.

Distillation Methods:

In [29] 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 [10]. 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 an adversarial feature, by blocking the defense method depends on to thwart attacks [10]. 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 various 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 [20] [35]. 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 [14]. See Figure 2.8 below for a typical architectural design of a honeypot system.

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 are honeypots deployed with the highest level of risk associated with them, this is in order to expose the full range of attacks initiated by

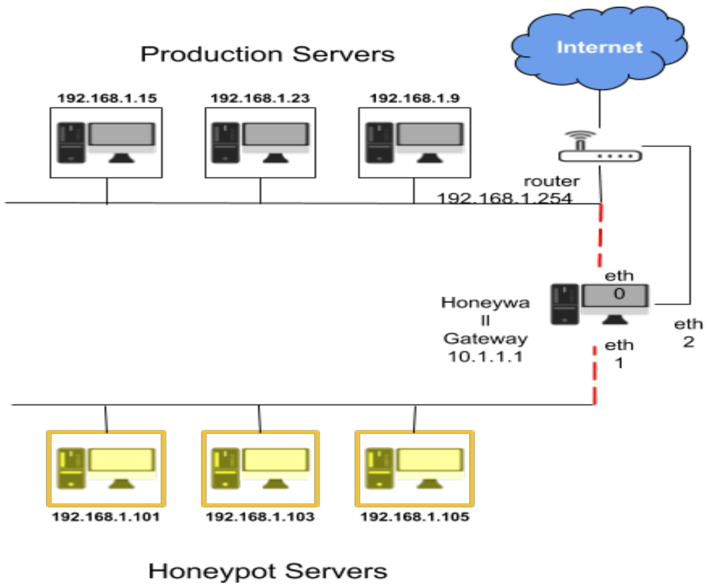


Figure 2.8: Classic Honeypot Architecture

the adversary. They are mainly used to collect statistical data on adversarial activities inside the honeypot [20]. They are 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 [23].

- **Production Honeypots** - they are honeypots known for ease of deployment and utility, and use in company production environment [23]. Closely monitored and maintained, their purpose lies in their ability to be used in an organization's security infrastructure to deflect probes and security attacks. They are attractive as an option for ease of deployment and the for sheer value of information collected on the adversary.

- **Physical/Virtual Honeypots** - physical honeypots are locally deployed honeypots, being part of the physical infrastructure. considered to be intricate and difficult to properly implement [20]. On the other hand, virtual honeypots are simulated systems (virtualized) by the host system to forward network traffic to the virtual honeypot [23].

- **Server/Client Honeypots** - 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 [23]. However, traditional honeypots are usually server-based.

- **Cloud Honeypots** - 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 [20].

- **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 *tokenize*, in order 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 [7]

2.4.3 Honeypot Deployment Modes

Honeypots can be deployed in one of three deployment modes [8], 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 Honeypot 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 is 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 in place. Here, we highlight a few [23]:

- **Valuable Data Collection** - honeypots collect data which are not polluted with noise from production activities and which are 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. The 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 a 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 (FP/FN).

Disadvantages of Honeypots

Ultimately, no security system or tool that exists is faultless. Honeypots suffers from some disadvantages, some of them are [23]:

- **Limited Field of View** - a honeypot is only useful if an adversary attacks it, 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 associated with it. 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 an 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 interaction the adversary has with the honeypot system environment [20]:

- **Low-Interaction Honeypot (LIHP) -** these type of system 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 2.9 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

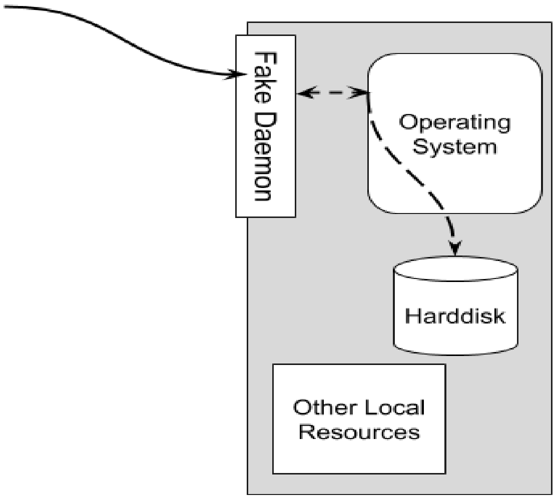


Figure 2.9: Classical Low-Interaction Honeypot Architecture

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 *Honeypy*.

- **High-Interaction Honeypot (HIHP) -** the main characteristic regarding High-Interaction Honeypots is that they are 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 2.10 below.

2.4.6 Uses of Honeypots

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) cyber-attack behavior, by analyzing incoming network traffic traversing through IoT nodes, and gathering attack intelligence [11]. In robotics, a honeypot was built to investigate remote network attacks on robotic systems [17]. Evidently, there is an increasing need to install *red herring* systems in place to thwart adversarial attacks before they occur, and cause damage to production systems.

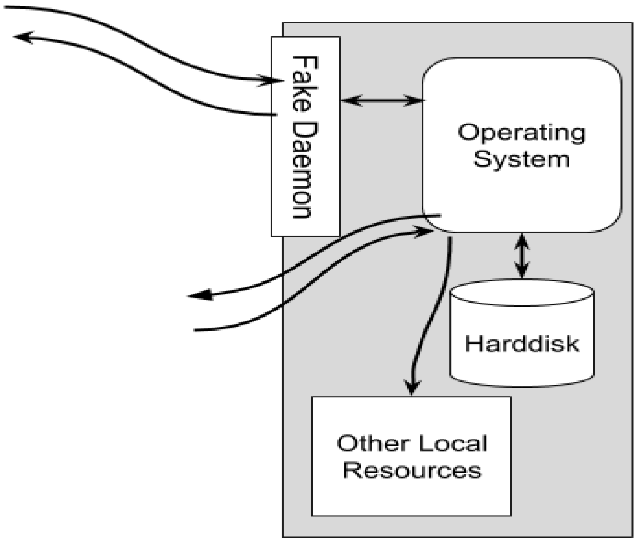


Figure 2.10: Classical High-Interaction Honeypot Architecture

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 can potentially capture the full spectrum of attacks launched by adversaries within the system. It 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 using *deception-as-a-defense*.

2.5 Honeypots 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 learning system, using honeypots as one of the primary components of deception in building the framework.

This decentralized framework must consist of H high-interaction honeypots. Each of these honeypots is embedded with a decoy target model T_{decoy}, designed to lure and prevent an adversary with adversarial input $\vec{x}$ from succeeding in causing a mislabeling attack $f(x) = y_{true}$ on the target model T_{target}. Essentially, the framework must perform the following tasks below.

Firstly, prevent the adversary from mimicking the neural network behavior in the learning function *f()* and replicating the decision space of the model. This will be done by blocking adversarial transferability, prevent the building of the correct substitute training model $F(S_p)$ from occurring and the transfer of samples from the substitute model F to the target model T_{target}. This makes it difficult to find a perturbation that satisfies $O\{\vec{x}+\delta\vec{x}\} = O\{\vec{x}\}$, since target model duplicated is fake.

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) TK_n, 2) making the network of honeypot nodes easily accessible 3) set up decoy target models T_{decoy}, deployed inside the honeypots for the attacker to interact with, instead of the actual target model T_{target}.

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 remote target model T_{target}, which in this case is a decoy; a decoy of the same optimization problem we saw in the earlier sections:

$$\vec{x^*} = \vec{x} + argmin\{\vec{z} : \hat{O}(\vec{x}+\vec{z}) \neq \hat{O}(\vec{x})\} = \vec{x} + \delta_x$$

This strenuous task is complicated further for the attacker because in order to generate the synthetic samples, the attacker must approximate the unknown target model architecture and structure F to train the substitute model $F(S_p)$, which is challenging. Evasion is further complicated as the number of deployed honeypots 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 computational time and resources. All the later, while keeping the actual target model T_{target} out of reach.

Chapter 3

Related Work

The purpose of this chapter is to summarize and evaluate earlier work in the literature on the techniques and frameworks designed to defend black-box learning systems from adversarial attacks. This chapter also covers discussions on how the *deception-as-a-defense* technique can used to protect security systems, specifically with the use of fake digital entities to attract, lure, and deceive adversaries.

The literature review below focuses directly on the concept of defending against adversarial examples, aimed at misleading the classifier. Most of the known defense methods are mainly based on data pre-processing and sanitation techniques, employed during the training phase of DNN model preparation. Pre-processing and sanitation typically mean influencing the effect that sample training-set data, X, has on neuron weights of the underlying DNN model, by distinguishing and filtering out malicious perturbations, inserted by an adversary that may mislead and/or confuse the classifier causing a misclassification or violation of model integrity. Other notable work in this section focus on the role of cyber-security defense through means of deception, specifically with the use 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 known adversarial defense techniques, but to supplement and reinforce existing ones, with the use of adversarial deception re-enforcing the application perimeter.

As an alternative, but conceptually close to the data pre-processing technique, is the use of *distillation as-a-defense* against adversarial perturbations [29], to reduce the impact impurities made on the neural net features. The authors use this simple technique to reduce the dimensionality of the DNN classification model, while maintaining accuracy. However, the work in [10] suggests that a model protected using defensive-distillation is no more secure than a model without no defense. While the latter method is able to defend against some cases of adversarial attack, it cannot protect the model's neural net against all types of attack, especially the ones that simultaneously target the different features, learned by the model.

The following is some of the research that deals with defenses against adversarial examples, or with *defense-through-deception* using digital breadcrumbs and tokens:

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 issue has been amplified by the fact that there is no unified understanding of how or what makes these DNN models so vulnerable to attacks caused by adversarial examples. The authors propose an effective solution which focuses on reinforcing the existent DNN model and making it robust against 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 a bounded ReLU activation function,$f_R(x) \rightarrow y$, in the DNN architecture to stabilize the overall model prediction ability. The second is based on augmented *Gaussian* data for training. Defenses based on data augmentation improve generalization since they consider both the true input and its perturbed version. The latter enables a broader range of searches in the input, then say adversarial training, which is limited in its partial of the input, causing it to fall short. The result of applying both strategies results in a much smoother and more stable model, without significantly degrading 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 the purpose of developing our proposed auxiliary defense technique, using honeypots. The authors in [15] propose a training approach aimed at building adversarial-resistant black-box learning systems against adversarial perturbations, by blocking transferability. The proposed method of training, called *NULL-labeling* works by evaluating input $\vec{x}$ and lowers confidence on the true label y, if $\vec{x}$ is suspected to be perturbed and rejecting it as invalid input. The criteria on which the method evaluates $\vec{x}$ is if it spans out of the training-data data distribution area. The training method smoothly labels, filters out, and discards invalid input (NULL), which does not resemble training-data. This is to prevent from allowing it to be classified into intended target label. The ingenuity of this approach lies in how it is able to decisively distinguish between clean and malicious input. NULL labeling proves its capability in blocking adversarial transferability and resisting the invalid input that attempts to 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 true label, all while maintaining prediction accuracy.

iii. *Towards Robust Deep Neural Networks with BANG* - this paper, [33], is another training approach for combating adversarial examples and fortifying the learning model. The authors propose this defense technique in response to adversarial examples, with their abnormal and ambiguous nature. The authors argue that model 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 has classified inputs correctly and has become resistant against malicious input perturbations that aim to exploiting and violating model integrity. This method is designed to avoid instability brought about by adversarial examples, which work by *pushing* the misclassified samples across the decision boundary into incorrect classes. This training method achieves good results on DNNs with two distinct datasets, and has low computational cost while maintaining classification accuracy for both sets.

iv. *HoneyCirculator: Distributing Credential HoneyToken for introspection of Web-Based Attack Cycle* - in this paper, [7], the authors suggest a framework that actively and purposefully leaks digital entities into the network to deceive adversaries and lure them to a honeypot that is covertly monitors, tracks token access, and records any new adversarial trends. In a period of one year, the monitored system was compromised by multiple adversaries, without being identified as a controlled decoy environment. 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 fake data to deceive, attract, and fool an adversary. The authors also 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. The defender's goal is to learn as much as possible about the attacker. The message that the authors try to convey is that as the threat of adversarial attacks increases, so will the need for novelty in the defense approaches to combat it.

v. *A Survey on Fake Entities as a Method to Detect and Monitor Malicious Activity* - this survey paper, [30], serves as an examination of the concept of *fake entities* and digital tokens, which my framework partially relies upon. 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, these inexpensive, lightweight, and easy-to-deploy *pawns* are as valuable as any of the other security mechanisms in the field, such as firewalls or a packet analyzers. Simply, they are digital objects, embedded with fake divulged information, intended to be found and accessed by the attacker. The authors advocate that operating-system based fake entities are the most attractive and fitting to become decoys, due to the variety of ways the operating system interface can be configured and customized. Once in possession of the attacker, the defender is notified and can begin monitoring the attacker's activity. Later in this work, the authors implement a framework that actively leaks credentials and leads adversaries to a controlled and monitored honeypot. However, the authors have yet to build a functioning 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 work that served as motivation for this thesis include: utilizing honeypots in defense techniques, such as design and implementation of a honey-trap [12]; deception in decentralized system environments [34]; and using containers in deceptive honeypots [18].

As mentioned at the end of *Chapter 2*, our approach using honeypots, does not seek to replace any of the existing methods to combat adversarial examples in a black-box attack context. However, it can be used effectively as an auxiliary method of protection that strengthen existing defense methods in production systems.

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 the context of a black-box attack setting. This proposed framework - *Adversarial Honeynet* lays a foundation for superimposing a defense system that *blankets* existing robust adversarial defense techniques, such adversarial training or distillation. At its core, it utilizes a computer security tool, known as *High-Interaction Honeypots (HIHP)*, 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 created by the attacker, as well as to prevent a targeted misclassification attack from violating the target model's integrity. Although the aforementioned framework increases defense costs if the attacker suspects a trap and aborts his exploits, there is still a high chance that an attacker will not disregard it and will still be drawn to it. The latter is one of the assumptions we have made while designing this system.

To the best of our knowledge, this is the first time the 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 with an adversary, through means of deception to block adversarial transferability from occurring, leading the adversary away from the target model T_{target} towards a decoy model T_{decoy} instead; 2) supplementing *less-than-secure* defense techniques by providing a fail-safe approach which works by augmenting existing security measures and enticing the adversary with elaborate deception techniques. This method does not add any extra complexity, but simply obstructs efforts of adversaries; 3) adversarial information reconnaissance through the use of high-interaction honeypots. The data collected can be used to analyze adversarial motives and techniques. Also, this proposed system is easily implementable, but due to time and resource constraints we were only able to implement one integral component, *Adversarial HoneyTokens* (please see *Chapter 5*).

This chapter starts with *Section 4.1*; a formulation of the problem definition that forms the basis of our thesis. We also highlight the design decisions and assumptions made prior to building this system in the sections that follows. *section 4.4* presents the threat model, which consists of the attack specificity, attacker capabilities, attack settings, and exploited vulnerabilities. *Section 4.5* and *Section 4.6* provides an overview and breakdown of the Adversarial Honeynet decentralized framework, wherein we discuss its functionality, architecture, and individual components. *Section 4.7* focuses on attracting the adversary to the honeypot through methods of deception. In *Section 4.8* we discuss how the framework monitors and detects malicious behavior. *Section 4.9* and *Section 4.10* explores how the adversary launches his attack and defends against an attack. Finally, *Section 4.11* focuses on the significance and novelty of our approach.

4.1 Problem Definition

To recap the thesis goal from *Chapter 1* of our thesis we want to determine whether a decentralized adversarial defense framework is possible to build, which utilizes high-interaction honeypots and several deception techniques, as well as operates within a black-box threat setting. This supplementary layer of defense protects learning systems from adversarial examples $\vec{x^*}$ that seek to destabilize the DNN learning model, by causing a targeted misclassification post training, jeopardizing the models integrity. However, the notion that adversarial examples simply destabilize is not well articulated, when the nature of this phenomena is more compelling and comprehensive.

These nonlinear input samples look familiar to the regular input samples $\vec{x}$, normally accepted by a DNN classifier, but they only superficially appear that way. If we recall, a black-box threat model is one where the adversary possess limited, if in fact no internal knowledge at all, of the learnings system architecture, structure, hyper-parameters, etc. One would assume a targeted attack to distort and compromise the acquired DNNs prediction ability through training would be unlikely, but that is obtuse without knowing all the details. This is because even with the little knowledge possessed by a potential adversary, a targeted attack in a black-box setting is still in fact probable. The latter is due to the foible concept known as *Adversarial Transferability*; which perpetuates its potency even in a black-box setting. From what we learned in *Chapter 2*, transferability states (in simple terms): that for an input sample $\vec{x}$, the adversarial examples $\vec{x^*}$ generated to confuse and mislead a target model m_1 can be transfered and used to target and confuse other models m_{n+1}, that are of homogeneous or even heterogeneous DNN architectures.

With the above postulate in mind, the adversary can build a substitute architecture F, with approximated model knowledge and synthetic labels collected S_p, by observing how the *Oracle* O labels the test samples $(x^{'}, y^{'})$. From what the adversary garners, he can build a substitute model F from what he learns from O, and train $F(S_o)$ that resembles the target model T_{target} in behavior. Now, with the ability to simulate and craft adversarial samples $\vec{x^*}$ that will be misclassified by the substitute model $F(S_p)$ and the target model T_{target}, when transferred. For as long as adversarial transferability principle holds, the same adversarial examples misclassified by *F()* substitute model will be misclassified by the target model. At its core, our intention is to devise a defense technique to both fool and prevent the attacker from interacting with the model. But before we begin to examine and breakdown the components of our proposed framework we must present some assumptions and design decisions we made during the deliberation process, which will help shape our defense solution.

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 knowledge of: 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, $q = \{q_1, q_2, q_3, ..., q_n\}$; 3) data features representation $f = \{f_1, f_2, f_3, ..., f_n\}$. We also assume the attacker has partial or no knowledge of the following, 1) purpose of DNN model; 2) DNN input and output layers of the DNN, represented by $X = \{X_1, X_2, X_3, ..., X_n\}$ and $Y = \{Y_1, Y_2, Y_3, ..., Y_n\}$ respectively; 3) existence of our honeypot (disguised as another 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 to 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 DoS 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 parallel connects/queries to more than one DNN decoy model T_{decoy_1}, T_{decoy_2} to build the substitute training-set S_ρ; 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] [22], which focus on curbing adversarial attacks by *normalizing* the input samples $\vec{x}$ injected into the target model T_{target}, our defense framework follows a different method of protection. 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 with and queries a decoy Oracle $\hat{O}$ 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 our 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

$$H_{adversarial} = \{H_1, H_2, ..., H_n\}$$

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 it 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 attacked 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 cautious and cunning in their methods.

Unwanted Congested Noise - As mentioned, this decentralized framework is designed to classify any system intrusion as a potential adversarial attack, since it is concealed. This leaves room for an increased rate of false positives. 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 violates 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 *Kibana* [**?**] 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 H 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_{target}, only *decoyed* replicas of the model T_{decoy}. The only disadvantage is that if 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) aggressively 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, heavy in computational spending.

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 interact with and exploit. The latter includes a real OS, real applications, real input/output and real services. But simulating a live system make it time consuming and complex to build, especially 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 other honeypots counterparts, 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, HIHPs were used in our work for the following 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 are limited in their data recording abilities. Also, high-interaction honeypots allows the defender to discover unknown adversarial strategies and exposed system vulnerabilities, such as *zero-day-attacks*. The latter is not possible in low and medium interaction types since they only simulate and give limited access to the OS services. These type of systems are not suited for our plan of defense.

Decentralized Systems vs Distributed Systems - 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, as it should be easy to exploit all the honeypots with the same attack strategy.

Use of a 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 is used for analysis, which analyzes the data from each individual Samba database component. The latter four tools work 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 a 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 [6].

Black-Box Models vs Blind Models - it has been clearly shown that an adversary can thwart defenses in both context settings, black-box and blind-models, as seen in [15] [29] [10] 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 to train his substitute classifier *F(S)* and transfer the examples to a target classifier T_{target}, to distort the classification of the output labels, as explained in *(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 observes 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 $(\acute{x}, \acute{y})$ to build the substitute model F and then generate the adversarial examples $\vec{x^*}$. 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 individual Samba databases. 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. Each adversarial honeypot node will store a moderate amount information, and since high-interaction honeypot only records/stores essential information about the session, we will 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 the 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 DoS 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 computational cost should the system scale-out, this something that must be discussed later in future work.

4.4 The Threat model

4.4.1 Attack Specificity

Generally, for an adversary to succeed in his attack, and whether the attacker has his sight set on violating the *availability* or *integrity* of the model, adversarial transferability needs to be successful. For purposes of our thesis, we have decided to design our adversarial attack to be a *targeted exploratory* one in nature [16]. A targeted attack is when the adversary has a specific set of data samples in mind, and is discriminatory in his attack. This means the adversary wants to force the DNN to output a specific target label y_{target}, $f(x) \longrightarrow y_{target}$, instead of the correct label y_{true}, $f(x) \nrightarrow y_{true}$. Please see Figure 4.1 below for an illustration of a adversarial targeted attack, violating model integrity. Hence, the adversarial examples generated need to have such an effect on the classifier, that it explicitly lowers the confidence on the target label. Misclassification attacks, to us, were less attractive since they do not make for interesting adversaries, not to mention the fact that these type of attacks appear random in nature, focusing on an arbitrary set of data samples. With no fringe inconsistencies to dispute, it becomes difficult to discern failures brought about by non-malicious factors effecting the classifier. Building on the latter, misclassification attacks make it all the more difficult to design defenses and robust frameworks to thwart adversaries when the attack itself seems arbitrary in nature.

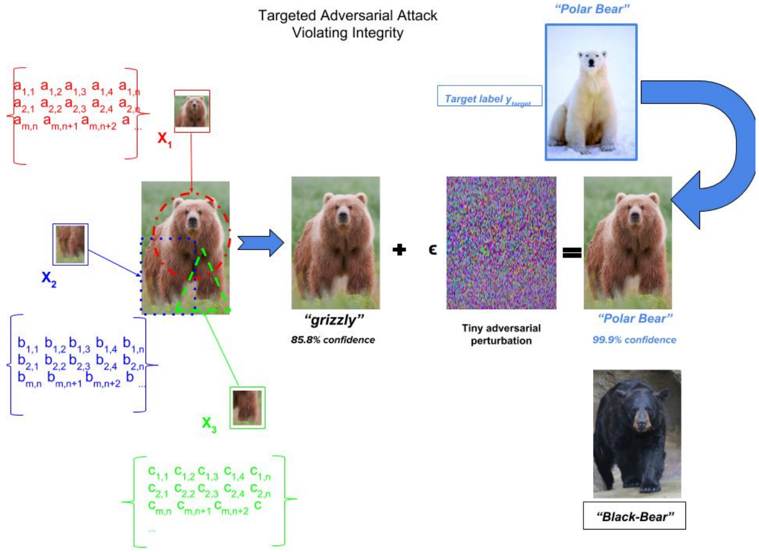

Targeted Adversarial Attack
Violating Integrity
"Polar Bear"
Target label y_{target}
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,n}$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,n}$
$a_{m,n}$ $a_{m,n+1}$ $a_{m,n+2}$ a ...
X_1
X_2
$b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{1,4}$ $b_{1,n}$
$b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $b_{2,4}$ $b_{2,n}$
$b_{m,n}$ $b_{m,n+1}$ $b_{m,n+2}$ b ...
X_3
$c_{1,1}$ $c_{1,2}$ $c_{1,3}$ $c_{1,4}$ $c_{1,n}$
$c_{2,1}$ $c_{2,2}$ $c_{2,3}$ $c_{2,4}$ $c_{2,n}$
$c_{m,n}$ $c_{m,n+1}$ $c_{m,n+2}$ c
...
+
ϵ
=
"grizzly"
85.8% confidence
Tiny adversarial perturbation
"Polar Bear"
99.9% confidence
"Black-Bear"

Figure 4.1: Adversarial Targeted Attack Violating; Input $\vec{x}$ (left) represented by different features $x_{1,2,3}$, perturbation $\vec{x} + \delta\vec{x}$ controlled by ε (middle), generated adversarial example $\vec{x^*}$(right). The adversary then forces the classifier to output the example with a different label, flipping it

4.4.2 Exploited Vulnerabilities

The cogent properties of adversarial examples $\vec{x^*}$ make them a prime candidate for adversarial attacks on deep learning systems. It should be anticipated that an ambitious and equally resourceful adversary will conspire to use these perturbations for malicious purposes. Generally, deep neural nets (DNN) work by extracting and learning the key multi-dimensional discriminate features $X_{m,n} = \{x_{n,1}, x_{n,2}, x_{n,3}, ..., x_{n,m}\}$ embedded within the input sample x pixels, to correctly classify it with the correct output label y_{true}. However, with adversarial examples entities, the acuity of a DNNs classification ability becomes slightly manipulable, and the adversary is aware of this weaknesses.

In our thesis, the designed adversary's attack depends on the successful exploitation of a fundamental vulnerability found in most, if not universally all DNN learning systems. This vulnerability is acquired during faulty model training. This weakness is embodied by a lack of non-linearity in poorly trained DNN models, that these visually indistinguishable adversarial examples, born in a high-dimensional space, epitomize. Other factors may also be responsible, such as poor model regularization. This inability to cope with non-linearity makes the DNN classifier insensitive to certain blind-spots in the high-dimensional classification region. Knowing the latter, an adversary can generate impressions of the input samples with slight perturbations. These examples can then be transferred between adjacent models, due to cross-model generalization property which allow the transfer of adversarial examples between the original and target model the adversary desires to exploit. The above vulnerability is manifested after the examples are synthesized and injected during the testing phase.

4.4.3 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 $q = \{q_1, q_2, q_3, ..., q_n\}$ is the exclusive and only capability an adversary possesses for learning about the target model and collecting his synthetic dataset S_p to build and gradually train his DNN substitute model F. See the Figure 4.2 below for an illustration of the adversary's capabilities, represented by the only capability, being able to query to *Oracle O*.

The adversary can create a small synthetic set of adversarial training samples from the

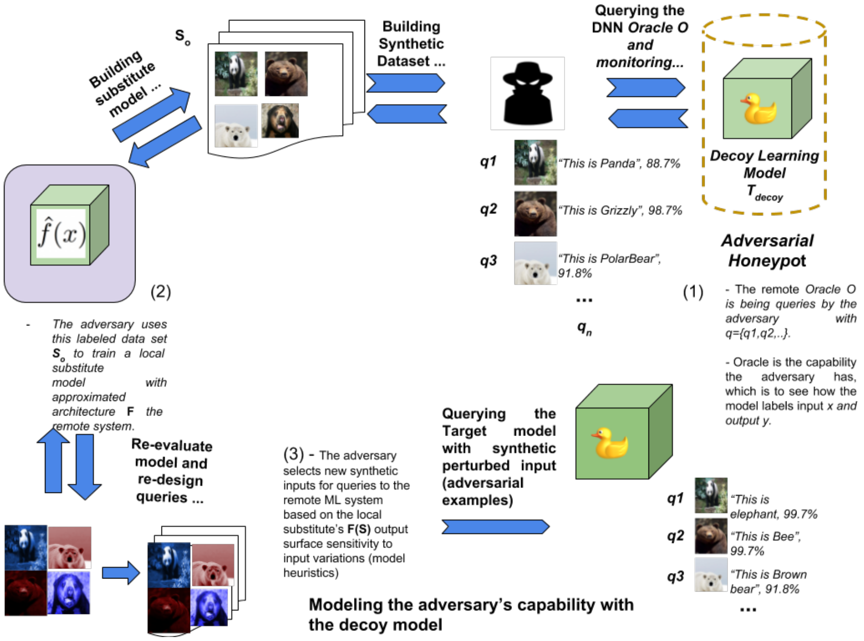

Building substitute model ...
S_o
Building Synthetic Dataset ...
Querying the DNN Oracle O and monitoring...
q1 "This is Panda", 88.7%
q2 "This is Grizzly", 98.7%
q3 "This is PolarBear", 91.8%
...
q_n
Decoy Learning Model T_{decoy}
Adversarial Honeypot
$\hat{f}(x)$
(2)
- The adversary uses this labeled data set S_o to train a local substitute model with approximated architecture F the remote system.
(1)
- The remote Oracle O is being queries by the adversary with q={q1,q2,..}.
- Oracle is the capability the adversary has, which is to see how the model labels input x and output y.
Re-evaluate model and re-design queries ...
(3) - The adversary selects new synthetic inputs for queries to the remote ML system based on the local substitute's F(S) output surface sensitivity to input variations (model heuristics)
Querying the Target model with synthetic perturbed input (adversarial examples)
q1 "This is elephant, 99.7%
q2 "This is Bee", 99.7%
q3 "This is Brown bear", 91.8%
...
Modeling the adversary's capability with the decoy model

Figure 4.2: Modeling the adversary's attack capability

initial set S_0 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 maps back to a given $x^{'}$ is the only capability that the attacker has 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. The 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 restrict himself to querying one model and will likely connect to multiple nodes and DNN model classifiers from the same connection for purposes of parallelizing synthetic data collection. This should trigger an alarm within our framework, indicating multiple access and that something abnormal is occurring.

4.4.4 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 F.

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_p 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 $\vec{x^*}$ 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 as 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 is challenging to design a model that can generalize against more advanced attacks, if not all. Also, it is 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 to re-train the network.

4.5 Adversarial Honeypot Network (Adversarial Honeynet)

4.5.1 Overview and General Architecture

The proposed *Adversarial Honeynet* framework is considered as an added layer of protection to *blanket* a deployed deep learning system, in order to combat imperceptible adversarial examples, within a black-box attack setting. There are several advantages and benefits that this framework can bring in the protection of existing learning systems. A single adversarial honeypot node in this decentralized framework may offer the following benefits: 1) *adversarial re-learning*; conceptually, it is a pragmatic method of collecting intelligence on the adversary, such as attack patterns, propagation, frequency and evolution. The latter results can be used to learn and reverse-engineer adversarial attacks; 2) an *anomalous classifier* used to identify whether the attackers actions are malicious or benign, this will help to determine whether or not to record the attacker's session information based on behavior patters against a *white-list*; 3) a *decoy target model*, used as a placeholder for the adversary to engage and interact in case his intention are indeed malicious in nature. The attacker's interaction with model is represented by the *Oracle* $\hat{O}$, that an adversary observes and queries, re-channeling his efforts; 4) an *Adversarial Database*, used to collect and securely store attack session data on the adversarys actions and maneuvers, used later to research and understand the adversary in *adversarial re-learning.*

All honeypot nodes are deployed with identical decoy models T_{decoy} that resemble the original target DNN model T_{target}. Also all services and applications on the high-interaction honeypot are real and not simulated, prompting the attacker to assume the model is indeed real, published or leaked by mistake. Neighboring adversarial honeypots are called *HoneyPeers*, these nodes are always active and have a weak non-privileged TCP/IP port open that is known to attract adversaries, spoored with adversarial honeytokens. The docker container node begins recording information when the anomalous classifier detects that the attacker is attempting to do something malicious and discretely notifies the neighboring *HoneyPeers* that an attacker is active within the network. *HoneyCollectors* are used to aggregate and collect information from each individual adversarial honeypot node and store it in the central *Adversarial Database.* All activities on the node are collected and stored with a public-key hashed *time-stamp.* In our framework, the central database is a *Samba* database is used to collect *structured,unstructured*, and *semi-structured* session data to record the *adversary-honeypot-decoy* interaction. An analysis module, used to aggregate adversarial information and use that to learn about the attacker, this learned information can potentially be used to perform inference for future attacks. See below for Figure 4.3 - an illustration of the Adversarial Honeynet architecture:

4.5.2 Functional System Components

- **HoneyPeers -** are a series of interconnected high-interaction honeypots joined in a decentralized network topology. Each HoneyPeer is an autonomous high-interaction honeypot contained node, with a copy of the decoy learning model T_{decoy}, embedded within a monitored Linux container, powered by Docker. Encrypted communication messages are passed between the nodes in order to notify adjacent nodes that an attack is occurring or has occurred. All communication is governed by our *message-passing-protocol* defined in *section (4.5.4).* Each *node-to-node* interaction is initiated by exchanging a *HoneySession Key*, which is used to authenticate a node's identity with each of its peers and is reused in verify future interactions. If a node should become unresponsive, it is assumed that the node has been *compromised and is infected.* In the

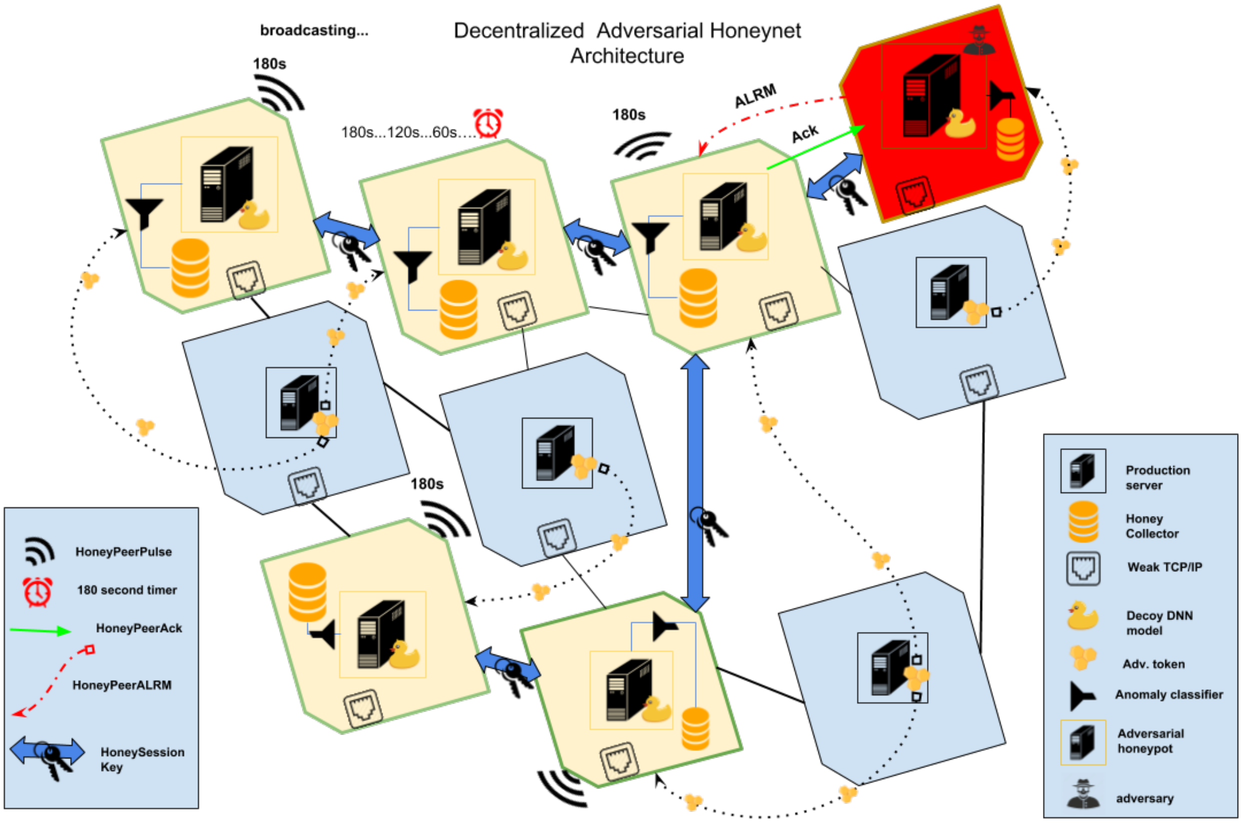


Figure 4.3: Adversarial Honeynet Architecture

case that a node should become infected, it can be assumed has been compromised by the adversary, in which case all neighboring nodes will severe all future communication with it, flag any local session HoneySession keys, and the infected honeypot will be cautionary labeled. Furthermore, all node-to-node interactions are securely stored and recorded in the *central adversarial database.*

- **Decoy Classifier -** represents our solution for preventing the adversary from interacting with the target classifier learning model T_{target}, and block *transferability* from occurring by re-channeling it to the honeypot. We distribute fake *decoy* learning systems throughout the enterprise or specifically in the anterior of a production system, acting as a type of *sentinel.* In our thesis, we hypothesize that legitimate users querying the learning system have no cause to interact with decoys or take notice of our adversarial honeypot. We decided to experiment with *deception-as-a-defense* using honeypot and decoys because we wanted to give the adversary a false sense of assurance, then identify and study them, and greatly reduce the rate of false-negatives FN violating classifier integrity.

  We suspect the adversary will attack our decoy learning classifier system T_{decoy} once he infiltrates the tailored honeypot container. It's purpose is to simply *simulate* and *mimic* value, in order to distract the adversary and prevent him from interacting with the legitimate target model T_{target}. If we consider the adversary to be *weak* with reference to (*section 4.2*), we see that the designed adversary only has partial knowledge of the model's purpose. This means the adversary does not have possess any internal details of the architecture, hidden layers, or hyper-parameters, etc. Knowing that the adversary is in a *black-box* setting and can only access input/output gives us great leverage over him. Before the adversary launches his attack, the adversarial actor in this case is like any other regular user in the system, with no systematic knowledge of the classifier.

Here, the adversary's capability to interact with the decoy model T_{decoy} is represented by the *Oracle* $\hat{O}$. $\hat{O}$ represents the means for an adversary to interact with and learn from decoy model. Since the adversary wishes to produce adversarial examples $\vec{x^*}$ for a specific set of input samples $\bar{x}$, collected by querying the $\hat{O}$, and then transfer them. However, adversarial transferability can be *re-channeled* if we can switch the target model T_{target} and the *Oracle* O with a decoy model T_{decoy} and thereupon *Oracle* $\hat{O}$, and convince the adversary that no *tampering* has occurred.

- **HoneyCollector -** is the component responsible for collecting all the adversarial session information on the adversary within each of the honeypot nodes in the network, it is the *Samba* component within our system.

- **Anomaly Classifier -** used to predict whether the adversary's actions inside the honeypot are considered abnormal or not. It depends on indicators, such as 1) *Number of DNN labeling requests*; 2) *execution of unusual scripts*; 3) *irregular outbound traffic from source*; 4) *sporadic DNN querying*; 5) *persistent activity on the DNN*; 6) *use of foreign synthetic data for labeling.*

- **Adversarial Tokens -** to summarize from *section 2.4.2*, it 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. More on this component will be discussed in *Chapter 5.*

4.5.3 HoneyPeer Node Inter-communication

This section describes the message passing protocol between the nodes in the adversarial Honeynet framework. A message can only be sent and received between two HoneyPeer nodes in the network that have exchanged *HoneySession* key between them. Any message that has been received or sent spontaneously should not be accepted. A reliable message passing technology must be set in place to avoid congestion and bottleneck at one of many parts of the network. Also, all messages sent, received, and dropped are *time-stamped* and recorded within the *adversarial central database* for bookkeeping purposes.

- **HoneyPeerALRM -** a distress message indicating that host node (*Sender*) has been compromised. The message is broadcast to the nearest adversarial honeypot node in the network. The neighboring nodes (*Receivers*) are responsible for intercepting and passing the message to all neighboring nodes in the network. For obvious security concerns and as fault-resistance, another HoneyPeerALRM message is sent on behalf of the *anomalous classifier*, in the case an adversary manages to seize control of the node and hijack it after detection. Each HoneyPeerALRM message must receive an *HoneyPeerACK* to indicate that the distress HoneyPeerALRM message has been received and acknowledged. Failure to reply might indicate one or several neighboring nodes have also been compromised. To add, nodes should not receive unsolicited HoneyPeerALRM reply messages from other adversarial nodes, as this may indicate malicious misrepresentation.

- **HoneyPeerAck -** this is a message sent corresponding to each HoneyPeerALRM message sent on behalf of the node. A HoneyPeerACK indicates that the distress HoneyPeerALRM message has been received and confirmed by the endpoint node. Failure to receive and acknowledge one ore more *Acks* might indicate that one or all the surrounding neighboring nodes have been compromised. Also, nodes should not receive unsolicited HoneyPeerALRM reply messages from other adversarial nodes.

- **HoneyPeerSafePulse -** Periodically, a honeypot node will send a *pulse* indicating that it is still active and part of the decentralized network, and not compromised. If the node neighboring it does not reply in 180 seconds with an *HoneyPotSafeAck* response, it is assumed that the node has been compromised.

- **HoneyPeerSafeAck -** A confirmation message sent to indicate that the node is active. After 3 consecutive (60 second interval) *no replies*, it can be assumed that either the receiving node is down or has been compromised, in which case, all neighboring nodes will severe all communication with it, purge any HoneySession keys, and the infected honeypot will be labeled as an *InfectedPeer*.

- **HoneySession Key -** An adversarial session key is exchanged between two HoneyPeer nodes. This HoneySession Key is exchanged at the beginning of a *node-to-node* interaction and will be used an authentication method in future *node-to-node* communications.

4.6 Individual Honeypot Node

4.6.1 Node Overview and Architecture

We can say the focal point of our adversarial Honeynet framework are the many deception nodes that comprise it. Each individual node castles a *high-interaction honeypot* (HIHP), complete with operating system services and resources that are made available to the adversary. In contrast to *medium interaction* and *low interaction honeypots*, nothing is simulated or fake. Also, here, the factor of realism lowers the risk of *fingerprinting.* Inside each of the deployed honeypots resides the decoy DNN model T_{decoy}, which the adversary will engage and exploit with his generated adversarial examples. However, with any deployed HIHP technology there is an inherent risk factor involved, one which cannot be ignored. Since all the service, applications and I/O are real and not simulated, there is a potential risk for leaving a vulnerable entry into the network. This means capturing and converting the vulnerable machine can be used to launch attacks from within the decentralized network. However, the great benefit here is that since all services and applications are real, the adversary is not inclined to suspect that this environment is indeed a trap. In order to maintain this cloak of deception the designer of this defense framework must find the appropriate balance between effectiveness of containment and the adversarial range of freedom within the honeypot. See below for Figure 4.4 which gives an illustration of the individual high-interaction node architecture:

4.6.2 Critical Components

- **Docker Image -** is the isolated *vessel* the encapsulates our decoy DNN model T_{decoy} and the software dependencies the DNN classifier depends on. A typical docker image is comprised of these components: 1) *writable container* which contains the binary files and external libraries; 2) a copy of the software or application; 3) a *base OS* which the application utilizes; 4) a *Kernel*, the Linux Kernel which manages the OS and all other programs, docker typically uses the kernel of the host OS it is built on.

- **Sysdig-Falco Component -** a rule-based behavior activity monitor designed to detect anomalous activity within *contained* applications [?]. The rules used filtering expressions. Falco, an add-on to Sysdig, used detect anomalous behavior and their

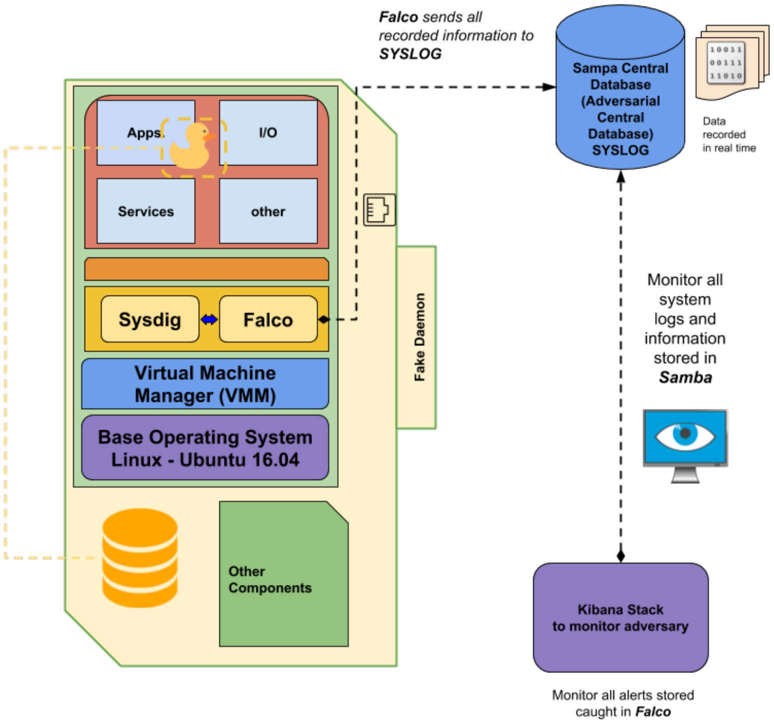

Falco sends all recorded information to SYSLOG
Sampa Central Database (Adversarial Central Database) SYSLOG
10011
00111
11010
Data recorded in real time
Apps
I/O
Services
other
Sysdig
Falco
Virtual Machine Manager (VMM)
Base Operating System Linux - Ubuntu 16.04
Fake Daemon
Other Components
Monitor all system logs and information stored in Samba
Kibana Stack to monitor adversary
Monitor all alerts stored caught in Falco

Figure 4.4: Individual Honeypot Component

arguments that occurs within the system. Sysdig-Falco is supported by Docker item **Kibana Component-** a browser based monitoring software which provides data visualization abilities for the content collected by the *Samba* database [?].

- **Samba Component -** an open source log recorder that records the anomalous and *sysdig* events [?]

4.7 Attracting The Adversary

4.7.1 Phase 1: Adversarial HoneyTokens

The first phase of our defense plan is soliciting the adversary. To re-channel the adversary's attention from the target model T_{target}, we need to provide an *alternative target* for him to focus his efforts on. The first-line in doing so is by planting fake digital information in the part of the production system where the adversary is most active.

Since we have dedicated an entire chapter towards implementing the adversarial honeytokens, we will not delve into this level of defense. Please see *Chapter 5* for all details on the adversarial honeytokens.

4.7.2 Phase 2: Accessible Honeypot

The subsequent phase of our defense plan is more logistical in nature. To attract the adversary to our network of *high-interaction* honeypots, we need to *invite* the adversary to take advantage and exploit the honeypot, by illicitly accessing it. The latter is challenging to resolve, since *beckoning* the adversary is subjective to the type of attacker we want to attract. The following are some of the considerations we took into account:

Non-privileged Network Ports - We assume that the adversary will be using network scanning software such as *scan* and *nmap*. In order to make our honeypot nodes easier to access, we intentionally leave vulnerable and well-known ports open for the adversary to access.

Use of Correct Bait - attributed to (*section 4.7.1*), the quality and location of adversarial tokens generated to lure the adversary is vital to the success of re-channeling the adversary's attack on the model. Regarding quality, though artificial and falsified in nature these *digital* pieces of information take any form or structure the creator chooses for it. Risk in lack of attacker interest is the reason for its use; to enhance the effectiveness of finding honeypots by the adversaries. This is why the token must be of high quality, to increase effectiveness, which is subjective in nature. Location is also important, placing the adversarial token in an ambiguous or hidden location might lowers the chances of its discovery by the adversary, which is why it is vital to place the tokens in a location or part of the production server frequented by the adversary.

Minimizing Risk of Fingerprinting - It is widely known that the fabricated information and resources inside a honeypot must resemble the real resource, *i.e*, T_{target}, the defender is tasks protecting. This is in order to avoid getting *fingerprinted*. According to section *(2.4.4)*, fingerprinting signifies the ability of the attacker to identify the presence of a honeypot. If the honeypot behaves differently than a real system, the adversary might identify and consequently detect it. And if its presence is detected, the attacker may simply ignore the honeypot and aggressively attack the entire network instead. In order to mitigate this risk, the environment where the artificial resource resides must seem realistic, i.e, *same look and feel* as a real production server , *same server type and version* as the production server, etc. However, it is extremely challenging to mimic the actual production server where the target model resides. However, in our case we are using *high-interaction* honeypots, which are preferred since it provides a real-live system for the attacker to be active in, nothing is artificial or simulated. However, there have been rare instances where the adversary was able to identify that environment was not real by identifying presence of VMware.

4.7.3 Phase 3: Decoy Target Model

The final and most important level of our *deception-as-a-defense* system is the decoy model T_{decoy}. Although unconventional, this model is supposed to delude the adversary into thinking he has accessed target model T_{target}. The goal of having the adversary interact with the T_{decoy} model is to engage the adversary in a hard optimization problem. The hard optimization problem is defined below.

Hard Optimization Problems

Ultimately, when the adversary accesses the T_{decoy} model embedded within the high-interaction honeypot, he possesses very little knowledge of its internal functionality. To produce the adversarial examples $\vec{x^*}$, he will need to observe the *Oracle* and gradually built a training set from $(x^{'}, y^{'})$ to train the substitute model F. The latter is dependent on successfully approximating an architecture for model F . What the adversary does not ration is that the training domain D he is collecting his samples $(x^{'}, y^{'})$ from to build his substitute model is collected from the a *decoy* target model T_{decoy} embedded by the defender, and not the legitimate model T_{target} accessed and queried by normal users. Unknowing to our set trap, the adversary collects the training set, gradually builds his substitute model form the synthetic dataset. With the least possible noise, represented by $\vec{x} + \delta\vec{x}$ and making it indistinguishable to humans, the adversary generates the adversarial examples $\vec{x^*}$. However, crafting these examples has a high computational cost overhead on the adversary since its expensive to generate the adversarial example. The latter is precisely what we use against the adversary. The targeted misclassification error the adversary is pursuing can only be achieved by crafting these perturbations with the least possible noise δ, to remain unnoticed. Finding the symmetry between the right amount of noise and sample indistinguishably signals an optimization problem, one that is *non-convex* and *non-linear* in nature. As mentioned in (*section 2.2.5*), a non-convex problem is one is where multiple solutions (local minimums) exist for the cost (loss) function $\min_x J_0(\theta)$ the adversary wishes to maximize. Theoretically, there is no one optimal guaranteed solution S that satisfies all, especially in high-dimensional adversarial data, see Figure 4.5 above for the adversarial solution surface. The adversary requires an exponential amount of time and variables to exhaustively find a feasible maxima solution.

The adversary original goal was to produce adversarial examples $\vec{x^*}$ for a specific input

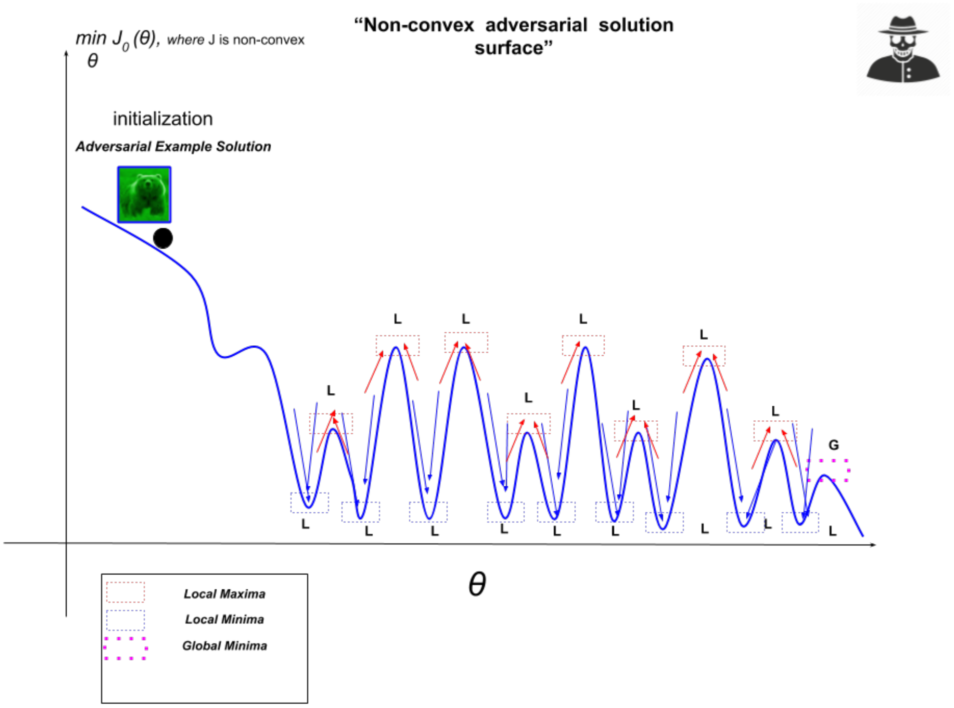

min J_0(θ), where J is non-convex
θ
“Non-convex adversarial solution surface”
initialization
Adversarial Example Solution
L
G
θ
Local Maxima
Local Minima
Global Minima

Figure 4.5: Non-convex adversarial solution surface. where multiple solution exist for the right allocation of perturbation

sample $\vec{x}$ to be misclassified by the *Oracle* O: $(T_{target}(\vec{x^*} = y) \neq (T_{target}(\vec{x} = y))$. This misclassification proves that the classifier has been compromised and is no longer usable. In our approach, we introduce *difficulty*, by deceiving the adversary and allowing him to attempt in solving this optimization problem (as he originally intended). However, these infeasible task for a decoy model T_{decoy} has no real value. Generating these adversarial examples is exhaustive in computational cost time, as well as approximating and training the substitute decoy model $F(S_0)$ to craft the examples. And if the attacker does indeed succeed in generating these examples, it would an highly infeasible task done in vanity.

4.8 Monitoring and Detecting the Adversary

One of the greatest challenges in our thesis was deciding how to adequately detect, classify and label adversarial behavior as malicious. Not to mention building the actual classification model that would be responsible for doing so would have been a great undertaking on its own. However, there were other practical detection methods at our disposal, such as using signature-based detection to compare an object's behavior against a *blacklist*, and anomaly-based detection to compare an object against a *white-list*. We chose to lean towards the former method (white-list) over blacklisting since we did not have reliable adversarial data that could have been used to generate a signature to fingerprint a potential adversary. White-list detection works best when attempting to detect entity behavior that falls out of anticipated and well-defined user actions, such as *over-querying the DNN model*, or *causing a sudden decline in the classification model performance*. White-list based anomaly detection fits perfectly into our defense framework since we can characterize any pattern of activities deviating from the norm as an intrusion. The latter is in our favor since we are trying to detect actions to exploit the classifier which are novel in nature.

Adversarial Behavior

In order detect adversarial anomaly behavior, we have surmised a list of adversarial actions and indicators that may signal *an-out-of the-ordinary* on the learning model. We will later use this indicators to build our white-list security rules. The following are some of those indicators:

- **Persistent DNN Querying -** while normal (non-adversaries) users will be querying the DNN T_{decoy} model with 1 or 2 queries per session, the adversary will be sending hundreds, if not thousands per session. All this in effort to build his synthetic training dataset S_p, the adversary will need to continuously collect training data, augment it and gradually train his substitute adversarial model *F(*S_0*)*. Repetitive queries $\tilde{Q}$ from the same source user within a set unit of time might indicate the adversary is *query-thrashing* the DNN model for labels $(x^{'}, y^{'})$. The latter could be a possible indication of adversarial attack on the prediction model.

- **Spontaneous DNN Activity -** In order for the adversary to craft adversarial examples $\vec{x^{*}}$, he will need to collect an initial set of labels S_0 from labeling $(x^{'}, y^{'})$. Then, he needs to build a substitute training model F that mimics the learning mechanism inherent in the decoy model T_{decoy}. naturally, collecting enough sample labels to accurately train the model F requires a large number of queries $\tilde{Q}$ solicited from the *Oracle* $\tilde{O}$. Consequently, in order to avoid raising suspicions, the adversary will try to build this initial substitute model training set S_0, as quickly and discretely as possible. The latter could be a possible indication of adversarial attack on the prediction model. This is true since a few queries is within normal user behavior, who have no malicious intent in mind. But spontaneously querying the oracle falls out of normal activity. See Figure 4.6 above for an illustration.

- **High number of DNN Labeling Requests -** an abnormally high number of query requests to the *Oracle* $\tilde{O}$ is not normal either. Let us not forget, that training of

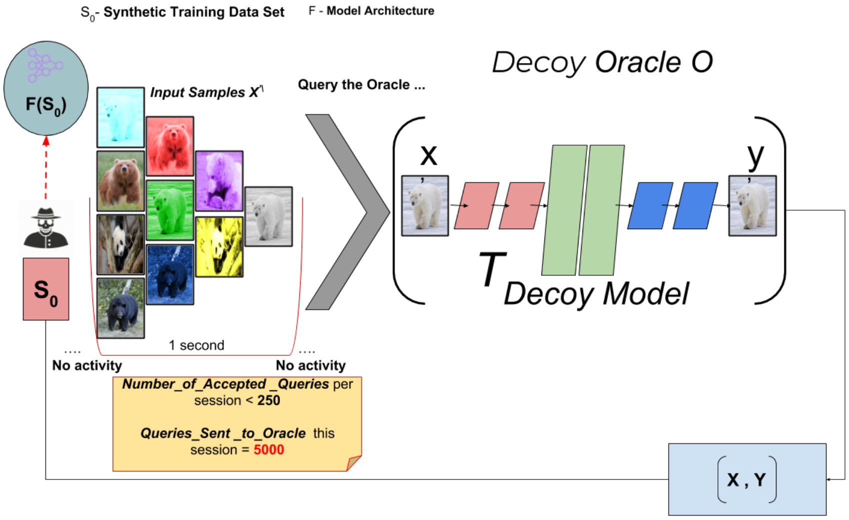


Figure 4.6: abnormal model activity indicating adversarial malicious behavior

the substitute model $F(S_0)$ is repeated several times in order to increase the DNN model accuracy and similarity to T_{decoy}. With each new substitute training epoch e, the adversary returns to $\tilde{O}$ and queries to augment (enlarge) the substitute model training set S_0 produced from labeling. This will produce a large training set with more synthetic data for training. With the correct model architecture F, the enlarged dataset is used to prototype the models decision boundaries separating the classification regions.

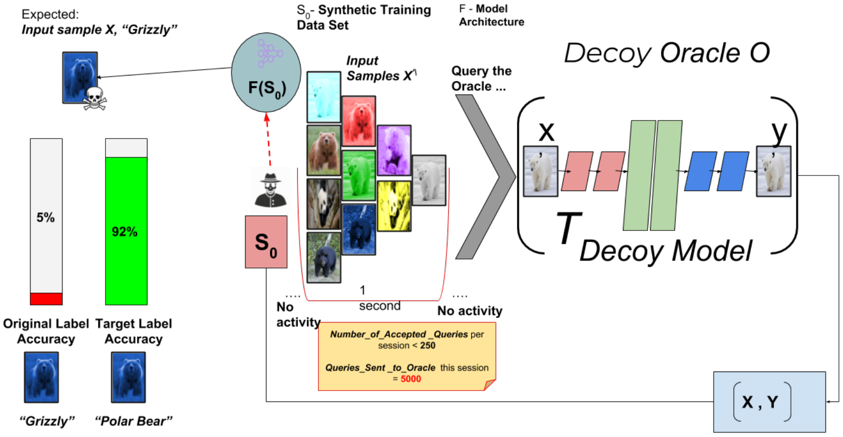


Figure 4.7: Targeted Attack Causes Decrease in Accuracy

- **Sudden Drop in Classification Accuracy -** Building on the above and as mentioned in (*Section 4.4.1*), our designed adversary seeks to cause a misclassification attack on the target decoy model T_{decoy}, by inserting malicious input(adversarial example, $\vec{x^*}$) in the testing phase. Because of this, an input unrecognizable to the model discriminate function 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. Other factors may influence a drop in accuracy, such as a poor learning or added bias in the data. This does not normally occur in a production environment, which indicates that our classification model is under attack. See Figure 4.7 for more details.

- other known indicator are more network related, such as *execution of unusual scripts alongside the DNN*, *Irregular outbound traffic or source*, *any sensitive or privileged path accessed during the interaction*, and *any spawning of suspicious child process.*

Detecting Malicious Behavior using Sysdig-Falco

Architecture: see Figure 4.8 below for an illustration of a typical architecture.

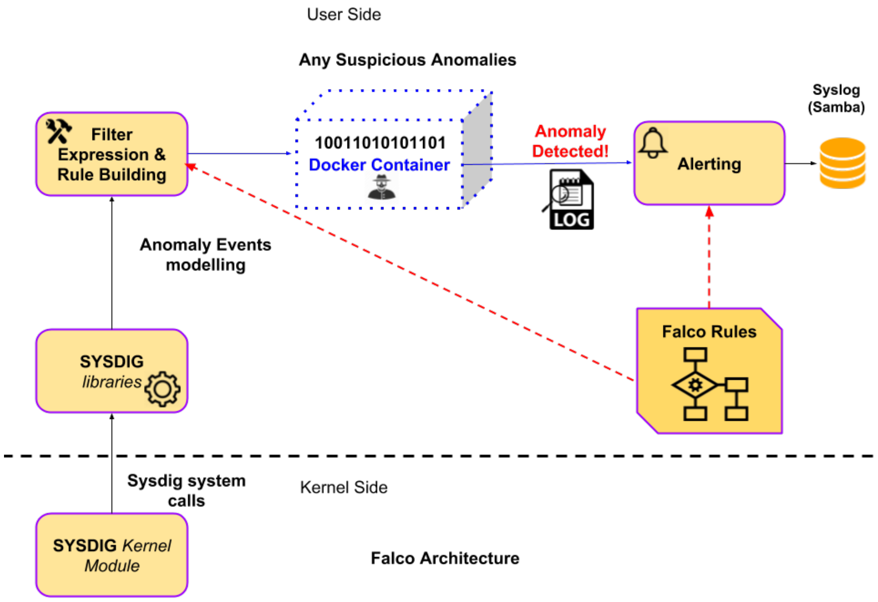


Figure 4.8: Sysdig-Falco Architecture

Components:

- **syslog listener (alterer) -** A syslog server needs to receive messages sent over the network. A listener process gathers syslog data sent using UDP protocol. UDP messages arent acknowledged or guaranteed to arrive, so some adversarial data maybe lost.

- **syslog Samba (database) -** each node within our large decentralized network can generate a huge amount of adversarial Syslog data. We decided to utilize a samba database to store and quickly retrieve adversarial syslog data.
- **syslog filter expression and rule building -** because of the potential for large amounts of adversarial data collected, it can be cumbersome to find specific log entries when needed. The solution is to use a syslog server that both automates part of the work, and makes it easy to filter and view important log messages. Syslog servers should be able to generate alerts, notifications, and alarms in response to select messages so that administrators will promptly know that an abnormality has occurred and can take swift action!

Configuration and Setting

- All abnormal events will be output in plain-text and securely stored within logs in the Samba (syslog) database.
- Adversarial *falco* monitoring rules are loaded startup initialization of the system.
- Security notifications and updates will not be output to the user in the Docker container. Instead, stored in a log file, whose contents are securely stored inside the Samba. database.

Security Rules -

Generally, there are 3 types of security rules: 1) Macro; 2) List; 3)Rules. For purposes of defining the white-list security rules in our thesis, we decided to define them using the *Rules* standard.

- Format The *Rules* security rules has the following format:

- *rule name* - the identifier of the rule.
- *desc* - a description of the rule, e.g. *rule for alerting on network traffic*".
- *condition* - a filter expression written in the simple Sysdig filtering language. It can contain macro components.

- *output* - the output message emitted when the rule is triggered. Written in Sysdigs output formatting. *Note: you can use fields from the event which will get interpolated.*
- *priority* - severity of rule (WARNING, INFO, etc.)

  Example:
  classifier-query-limit(rule name).
  an attempt to send more then 250 queries per the session limit(desc).
  query-limit $\leqq$ *250*(condition).
  The user has exceeded the set query limit per session(output)
  ERROR(priority)

Limitations -

There are some limitations to utilizing the *syslog-falco* architecture in our framework:

- *Falco rule format inconsistency* - syslog falco provides a customized way of formatting an output message emitted when the rule is triggered by the adversary's action. The latter proves problematic, since different *sysdig-falco* architectures will be set up by different defenders, hence set with various output message formats. Some messages will be clear, while some aren't. While this data will be used to learn from the adversary's interaction with the target model, it will have to be standardized in all honeypot nodes.
- *UDP transport protocol* - syslog-falco architecture uses UDP transport protocol to transport messages and then store them. However, it is widely known that UDP protocol is unreliable due to packet loss and overall network congestion. This may lead to the loss of some messages between the nodes. Switching to a more reliable message passing method, such as *Kafka* or *MQS* might be a better option.
- *No centralized way to collect data* - aggregating the reconnaissance from each docker container in each honeypot is the sole responsibility of each honeypot node to do so. Since our framework is a decentralized one, there is no parent node or centralized collector to collect data, this is in order to avoid a single point of failure. All data aggregations are integrated separately.
- *White-list rules are stored inside a file* - syslog-falco only provides the core components of what is needed to monitor and detect adversarial anomaly behavior. This means there is no access-control protocol or verification process in place to limit access to the *white-list* rule file. if an adversary were to detect a trap and aggressively seek to

hijack the honeypot, he could simply access the white-list file and modify the rules to disable the pitfalls designed to monitor him. This suggests the defender should devise a method to lock unauthorized access to the *white-list* rule file.

- *Log files are fairly large* - the syslog files generated by collecting anomaly related information on the adversary are fairly large and too complex to comprehend, most of all manually. It might prove difficult, with an automation process, to catalog all the information collected for purposes of adversarial re-learning.

4.9 Launching the Attack

In this section, we describe the typical adversarial synopsis our attacker will orchestrate to inject adversarial examples $\vec{x^*}$ into the the target model T_{target} during the testing phase. This culminates in a targeted misclassification $T_{target}(\vec{x}) \neq y_{true}$. As mentioned, this attack occurs within a black-box system context, is characterized by the following steps:

4.9.1 Accessing the Honeypot

We assume at this instance, that the adversary has successfully maneuvered himself and accessed one of the numerous high-interaction adversarial honeypot node systems H_1 in the set of nodes $H_{adversarial} = H_1, H_2, H_3, ..., H_n$, actively deployed within our decentralized network. We assume that:

- the adversary has scanned (*via scan, nmap, etc*), identified and accessed one of the honeypot node system, through intentional illegal access of one the many vulnerable non-privileged network ports that we have kept accessible (refer to *section 4.7.2* for more details).

- we have successfully gained the adversary's attention and trust, sufficiently enough to re-channel his interest away from the production systems. With the correct qualitative adversarial deception honeytokens *Token* planted in a strategic location, the adversary has shifted focus fo attack. We assume that the adversary's attention is now focused on and interacting with the T_{decoy} in the H_1, which he has conveniently found deployed in the honeypot (refer to *section 4.7.2* for more details).

- There is no risk of the adversary fingerprinting our decoy system honeypot H_1 true identity as a decoy trap. The adversary presumes that the high-interaction H_1 system resources are real, and the decoy target model T_{decoy} has been legitimately leaked, by virtue of negligence (refer to *section 4.7.2* for more details).

4.9.2 Initiating the Attack

As mentioned in the attack setting (see *section 4.4.4*), our adversary is now interacting with a black-box learning system. As per our assumptions, the adversary does not possess any internal vital knowledge regarding the core functional components of the decoy target DNN model T_{decoy}. Our adversary is *weak*, limited by what he knows, with no assumed knowledge of *hidden layer architecture*, *model hyper parameters*, *learning rate*, etc (see *section 4.2*), see Figure 4.9 below. He has limited knowledge of the features dataset $f = f_1, f_2, f_3, ..., fn$, has only the ability to effectively monitor the surrogate decoy target model T_{decoy} through means of querying its implicit *Oracle* $\hat{O}$ (outlined in the first step of the attacker's capabilities - *section 4.4.3*). In the next step, we see how the adversary approximates and selects a model architecture F to train the substitute training model.

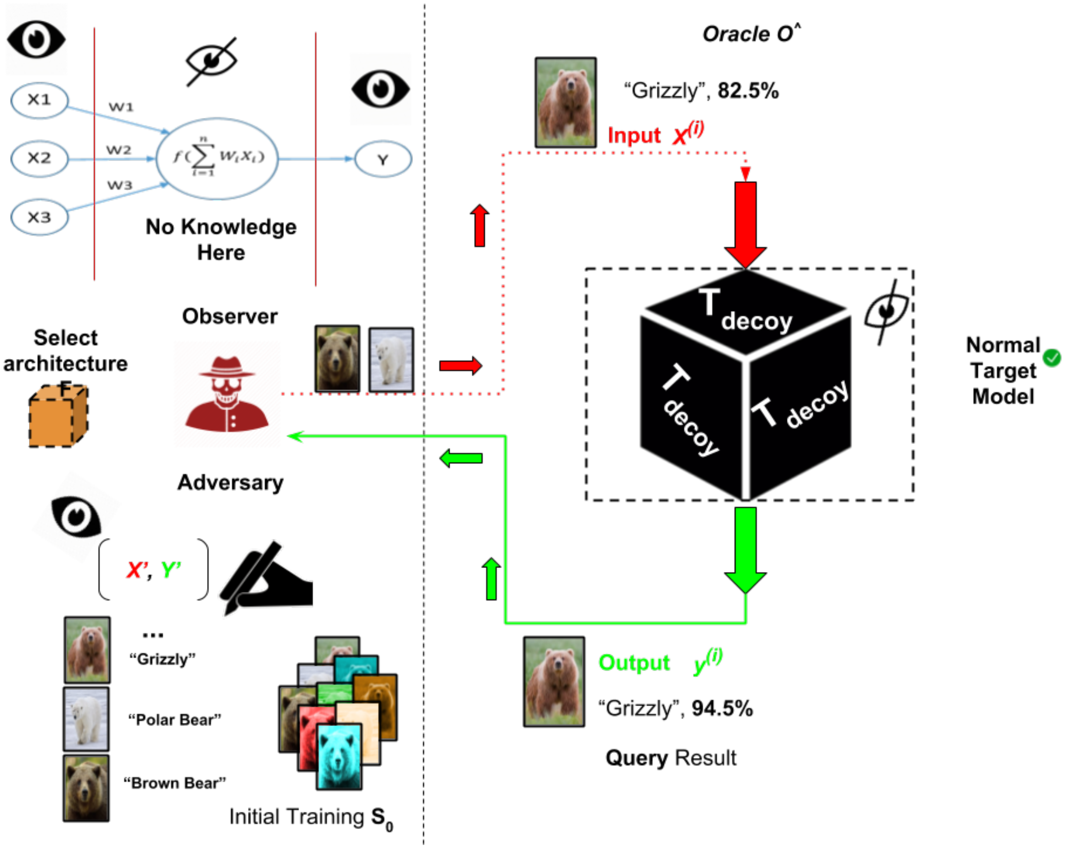


Figure 4.9: Initiating the Attack and Building the Initial Training set

4.9.3 Defining the Architecture *F*

This first step is considered crucial in constructing the substitute training model to craft adversarial examples. This is because the adversary must experiment with different substitute model architectures $F_{1..n}$ until he finds the most appropriate one. This is due to the notion that without knowledge of the target decoy model T_{decoy} architecture, the attacker knows very little about how the system learns, processes input $\vec{x}$ (text, images, or any media), and produces output y (label or probability vector). One way is to explore and experiment with different ones, until we find one that forces a misclassification on the target label y_{target} with successful results, which is what the authors did in the designed attack in [27].

Algorithm 2 - Substitute DNN Training: for oracle $\hat{O}$, a maximum number max_p of substitute training epochs, a substitute architecture F, and an initial training set S_0.

```
Input: $\hat{O}, max_{\rho-1}, S_\rho, \lambda$
Define architecture F
for $\rho \in 0...max_{\rho-1}$ do
        //  Label the substitute training set
    $D \leftarrow \{(\vec{x}, \hat{O}(\vec{x})) : \vec{x} \in S_\rho\}$
        //  Train F on D to evaluate parameter $\theta_F$
    $\theta_F \leftarrow train(F, D)$
        //  Perform Jacobian-based dataset augmentation
    $S_{\rho+1} \leftarrow \{\vec{x} + \lambda \cdot sign(J_F[\widehat{O(\vec{x})}]) : \vec{x} \in S_\rho\} \cup S_\rho$
end for
return $\theta_F$
```

4.9.4 Labeling the Initial Training Set S_0

At this stage, the adversary begins amassing a training set S_0, manufactured by using the small test sample $(x^{'}, y^{'})$ set pairs at his disposal. The collected test samples are used to query *Oracle* $\hat{O}$ of the hidden decoy, observe and record its labeling behavior $\{(\vec{x}, \hat{O}(\vec{x})) : \vec{x} \in S_p\}$. This assembled training set S_0 shares many of the statistical properties as the target model data domain *D*, such as data distribution and variance. The initial training set S_0 will continue to expand through the maximum number *epoch* cycles max_p, as the training model slowly converges.

4.9.5 Training the Substitute Model $F(S_0)$

The adversary gradually trains and updates the weights θ of substitute training model $F(S_0)$. Using a continually updated set D set(collected from the step above) the model F is trained, $\theta_F \longleftarrow train(F, D)$, see Figure 4.10 above. In order to expand the training set, a data augmentation technique is used on the updated training set S_p in order to produce a much larger training set S_{p+1}, with more training data points. *Labeling* and *Substitute training* are

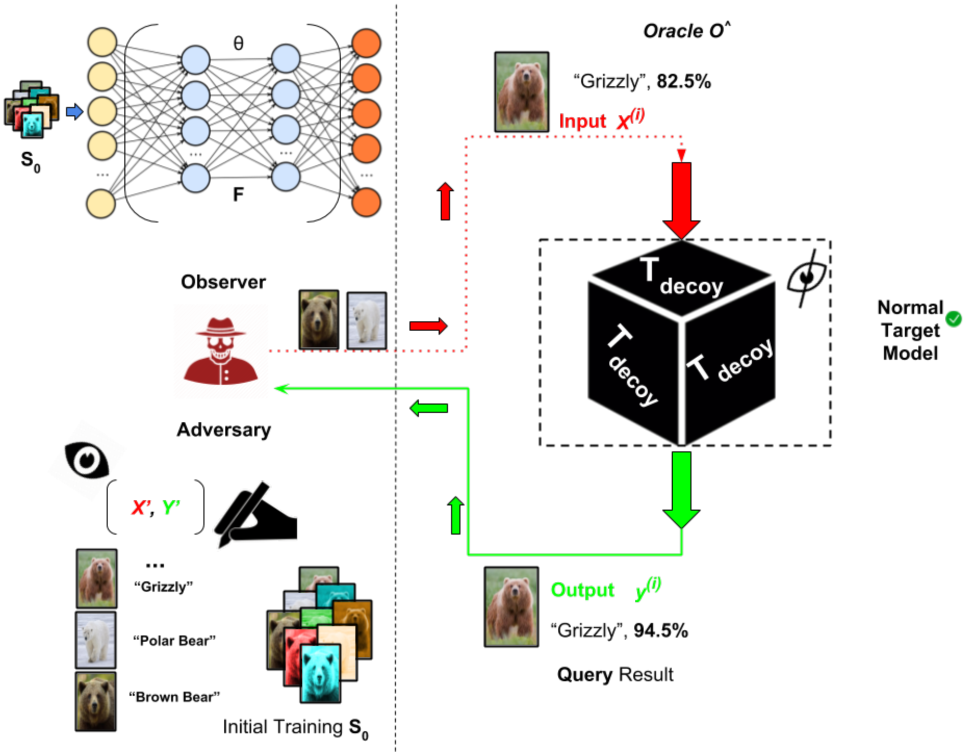


Figure 4.10: Training the Substitute Model with Initial Set

repeated for the augmented dataset. This continues several times in order to increase the adversarial substitute model F accuracy, until it is satisfactory and mimics the decision boundaries of the decoy target *Oracle* $\hat{O}$. The augmentation method is *Jacobian-based dataset augmentation* where the new training division S_{p+1} by $\{\vec{x} + \lambda \cdot sign(J_F[\hat{O}(\vec{x})]) : \vec{x} \in S_p\} \cup S_p$. Here, λ is the parameter of augmentation, which represents the step taken in the direction to augment from S_p to S_{p+1}, see Figure 4.11 for an illustration.

4.9.6 Generating the Example $\vec{x}^*$

At this point, the adversary has trained his substitute DNN training model, now it is ready to craft adversarial examples $\vec{x}^*$. For purposes of generating the adversarial examples, we have decided to use the *Papernot Algorithm*, titled *Jacobian-based Salience Map Approach* (JSMA), referenced in (*section 2.2.4*). The reason for selecting JSMA is because we wanted to design our adversary with a disadvantage, as this method requires less added perturbations $\delta\vec{x}$, but requires greater computational cost, which is an advantage for the defender.

Compute the Jacobian Matrix - First, the adversary computes the *Jacobian Matrix* $J(f)(\vec{x}) \in R^{(m \times n)}$ for the input $\vec{x}$ the adversary wishes to create an adversarial example of,

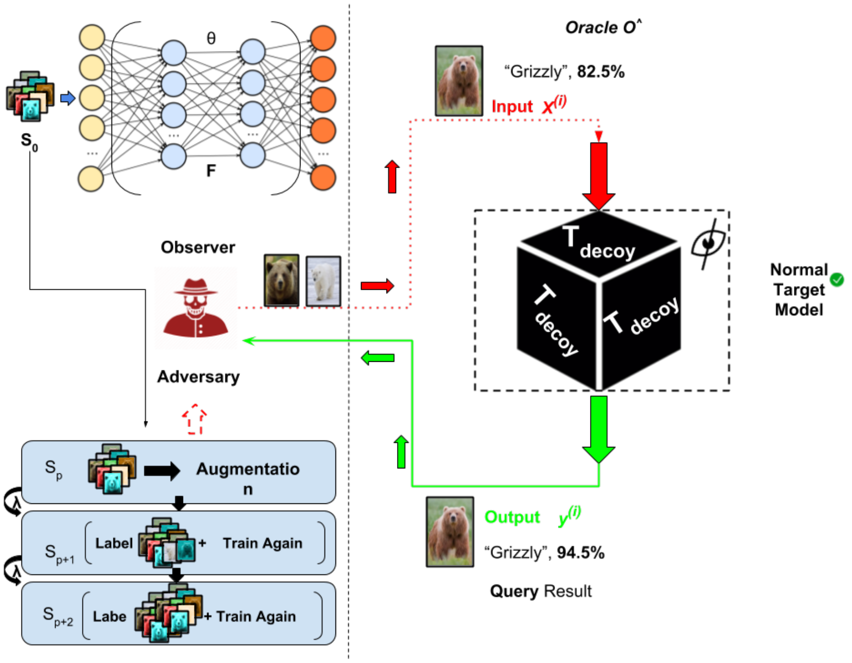


Figure 4.11: Data Augmentation to build the training set Sp to Sp+1

$\vec{x}^*$. The result of which are first-order derivatives $f'(\vec{x})$ of $\vec{x}$. From this, the adversary gets a matrix $i \times j$ 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 label j with respect to feature i in the input $\vec{x}$.

Find the Saliency Map of each Input Feature - The adversary then computes the *saliency map*, and computes the score S for each input feature matrix item i for input $\vec{x}$, given to us by [25].

$$S(\vec{x},t)[i] = \begin{cases} 0 \text{ if } \quad \frac{\partial f_i(\vec{x})}{\partial \vec{x_i}} < 0 \quad \text{or} \quad \sum_{j\neq t} \frac{\partial f_i(\vec{x})}{\partial \vec{x_i}} > 0 \\ (\frac{\partial f_i(\vec{x})}{\partial \vec{x_i}}) \quad |\sum_{j\neq t} \frac{\partial f_i(\vec{x})}{\partial \vec{x_i}}| \quad \text{otherwise} \end{cases} \tag{4.1}$$

The value t is the target class label y_{target}, that we wish to assign the input sample $\vec{x}$ instead of the source class label y_{true}, to fulfill our misclassification goal. The Jacobian Matrix here is $\frac{\delta f_j \vec{x}}{\delta \vec{x_i}}$. See Figure 4.12 below for an illustration.

Maximize Loss - The adversary then selects from the a sorted list of decreasing saliency value item i. Then one by one, The adversary adds the component i, where $S(\vec{x},t)[i]$ to the perturbation of the sample $\delta\vec{x} + \vec{x}$. The latter process is repeated several times until the adversarial sample $\vec{x}^*$ causes a misclassification in the substitute model F (see *section 4.9.7*).

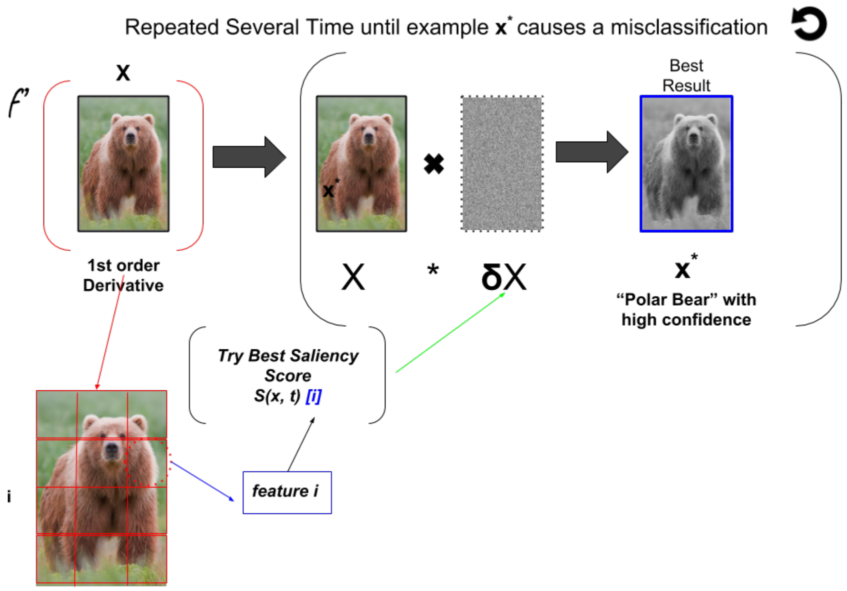

Repeated Several Time until example x* causes a misclassification
f'
x
1st order Derivative
x*
Best Result
X
*
δX
x*
“Polar Bear” with high confidence
Try Best Saliency Score S(x, t) [i]
feature i
i

Figure 4.12: Jacobian Saliency Map Approach to Generate Adversarial Example

4.9.7 Example Transfer $\vec{x}^*$

As mentioned in (*section 2.3.5*), It was proposed that transferability can be used to transfer adversarial examples from one DNN model to another, that share a common purpose or task, yet are dissimilar in network architecture. Here, we see that transferability is essential for adversary's black-box attacks on the target decoy model T_{decoy}. With the adversarial example generated above $\vec{x}^*$, adversary with the aid of the transferability property can launch a targeted classification attack. The adversary has trained a substitute model $F(S_p)$, generated transferable examples $\vec{x}^*$ using JSMA, and can now transfer them to the deployed target model, misclassifying the victim's trained model. See Figure 4.13 below for an illustration.

4.9.8 Flipping the Target Label

Once the adversary has his adversarial example ready, it is time to inject it into the decoy target model T_{decoy} at test time, assuming adversarial transferability holds.

At this stage, the adversary has all he needs to succeed in his targeted attack. The adversary has decided to design his adversarial attack to be a *targeted exploratory* one in nature. The adversarial example selected $\vec{x}^*$ is *infused* with a perturbation $\vec{x} + \delta \cdot \vec{x}$, which he will maximize the error on the false negative *(FN)* in the classifier. This means the adversary will exploit the vulnerability mentioned in (*section 4.4.2*) to force the DNN to output a specific target label y_{target}. Hence, the adversarial examples $\vec{x}^*$ generated using

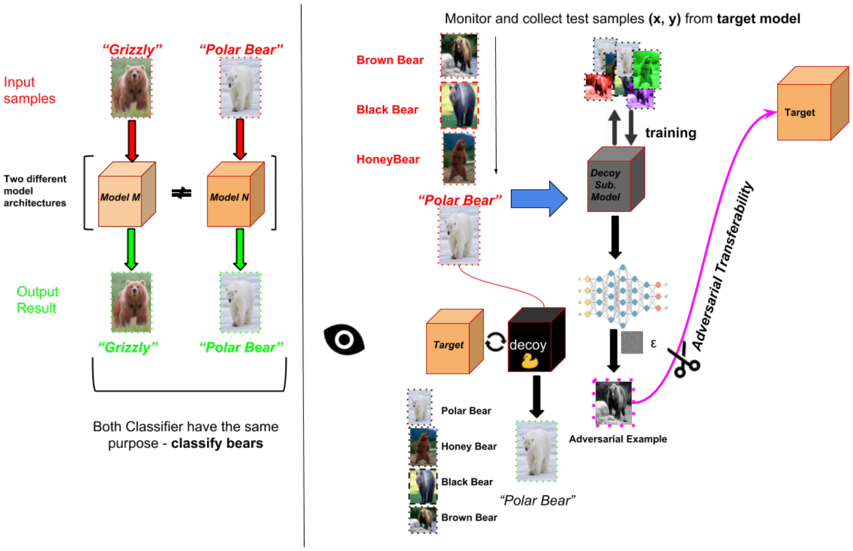


Figure 4.13: Adversarial Transferability

JSMA need to have such an effect on the decoy target classifier T_{decoy}, that it explicitly lowers the confidence on the true label y_{true}.

4.10 Defending Against the Attack

In this section, we describe the typical adversarial defense our adversarial Honeynet framework will provide to supplement existing adversarial defense systems. This is in order to combat malicious adversarial examples $\vec{x^*}$ and unwanted adversaries, during the deployment of a target model T_{target} we wish to protect. With this, we hope to thwart, contain and learn from the adversary with a first-line-of-defense, culminating with the goal of blocking adversarial transferability from learning the true behavior of the target model T_{target}. The defense is characterized by the following steps:

4.10.1 Luring and Baiting the Adversary

We assume at this instance, that the adversary has discovered, accessed and been deceived by one or two of the adversarial honeytokens $TK = \{TK_1, TK_2, TK_3, ..., TK_n\}$ generated and deployed in one the production systems. The adversary is now in the process of locating one of the numerous high-interaction adversarial honeypot node systems H_1 in the set of nodes $H_{adversarial} = \{H_1, H_2, H_3, ..., H_n\}$, actively deployed within our decentralized network. See *Chapter 5* for a more detailed implementation

4.10.2 Inside the Honeypot

Inside the adversarial honeypot $H_{adversarial}$, the adversary operates within a high-interaction virtual environment with OS real applications, services and devices. Any action the adversary takes, malicious or benign is monitored and stored inside the *Samba-Sysdig* database. Within the honeypot itself, several defense mechanisms have been put in place to thwart any unhanded attempts by the adversary, consider the following:

Message-Passing to Prevent any Adversarial Takeover: All HoneyPeer nodes in the decentralized network are in constant inter-communication between each other. Encrypted messages are passed between the nodes in order to notify adjacent nodes that an adversarial attack is occurring or has occurred. These time-stamped messages can only be sent between two nodes in the network, which have exchanged a honey-session between them. Explained in *section 4.5.4*, these messages include the *HoneyPeerALRM, HoneyPeerAck, HoneyPeerSafePulse*, and *HoneyPeerSafeAck*. See Figure 4.3 above for more details.

HoneySession Key to Authenticate Node Identities: An adversarial session key is initial exchanged between two HoneyPeer nodes. This HoneySession Key is exchanged at the beginning of a *node-to-node* interaction and will be used an authentication method in future *node-to-node* communications.

Anomaly-based Detection Using a White-list: White-list based anomaly detection used perfectly into our defense framework to characterize any pattern of activities deviating from the norm as an intrusion. This latter is in our favor since we are trying to detect actions to exploit the classifier which are novel in nature, such as *zero-day-attacks*.

4.10.3 Hard Optimization Problem

Inside the VM, when the adversary accesses the T_{decoy} model embedded within the high-interaction honeypot, he possesses very little knowledge of its internal functionality. To produce the adversarial examples $\vec{x^*}$, he will need to observe the *Oracle* and gradually built set of training samples $(x^{'}, y^{'})$ in order to build the training set for the substitute model F. The latter is dependent on successfully approximating an architecture for model F . What the adversary does not ration is that the training domain D he is collecting his samples $(x^{'}, y^{'})$ from to build his substitute model is collected from the a *decoy* target model T_{decoy} embedded by the defender, and not the legitimate model T_{target} accessed and queried by normal users. Unknowing to our set trap, the adversary collects the training set, gradually builds his substitute model from the synthetic dataset. With the least possible noise, represented by $\vec{x} + \delta\vec{x}$ and making it indistinguishable to humans, the adversary generates the adversarial examples $\vec{x^*}$. However, crafting these examples has a high computational cost overhead on the adversary. The latter is precisely what we use against the adversary. The targeted misclassification error the adversary is pursuing can only be achieved by crafting these perturbations with the least possible noise δ, to remain unnoticed. Finding the symmetry between the right amount of noise and sample indistinguishably signals an optimization problem, one that is *non-convex* and *non-linear* in nature. As mentioned in *section 2.2.5*, a non-convex problem is one is where multiple solutions (local minimums) exist for the cost (loss) function $\min_x J_0(\theta)$ the adversary wishes to maximize. Theoretically, there is no one optimal guaranteed solution S that satisfies all, especially in high-dimensional adversarial data, see Figure 4.5 above for the adversarial solution surface. The adversary requires an exponential amount of time and variables to exhaustively find a feasible maxima solution.

The adversary original goal was to produce adversarial examples $\vec{x^*}$ for a specific input sample $\vec{x}$ to be misclassified by the *Oracle O*: $(T_{target}(\vec{x^*} = y) \neq (T_{target}(\vec{x} = y))$. This misclassification proves that the classifier has been compromised and is no longer usable. In our approach, we introduce *difficulty*, by deceiving the adversary and allowing him to attempt in solving this optimization problem (as he originally intended). However, these infeasible task for a decoy model T_{decoy} has no real value. Generating these adversarial examples is exhaustive in computational cost time, as well as approximating and training the substitute decoy model $F(S_0)$ to craft the examples. And if the attacker does indeed succeed in generating these examples, it would an highly infeasible task done in vanity.

4.11 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 $(x^{'}, y^{'})$ 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 [13] and Defensive Distillation [10] [29] 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_{decoy}. 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.

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 [6] repository to generate tokens pertaining to the DNN classifier and use the Linux based tool *Auditd* to monitor token access.

- *Honeypot Accessibility (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 is done by taking the following considerations, keeping a non-privileged network port open, use of correct token bait, and minimizing risk of fingerprinting. We assume the attacker is using port scanning software such as *Nmap* or *Nessus* tools.

- *decoy DNN model - (between attacker and environment)* The final and most important level of our *deception-as-a-defense* system is the decoy model T_{decoy}. Although unconventional, this model is supposed to delude the adversary into thinking he has accessed target model T_{target}. The goal of this is having the adversary interact with the T_{decoy} model in order to engage the adversary in a hard optimization problem, with no profitable outcome.

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 [6]. Firstly, in (*section 5.1*) we provide a background to these tokens, as well as explain the nature and purpose of the honeytoken bits. Then in (*section 5.2*), we provide an outline of the individual project components and the project's hierarchical structure. The rest of this chapter focuses on the software architecture and design (*section 5.3*), functional features (*section 5.4*), functionality (*section 5.5*), usage, deployment scenarios and strategies (*section 5.6*). We then end this chapter with a highlight of the external dependencies (*section 5.7*) which our extension - *Adversarial Honeytokens* and its predecessor depends on, integration, and benefits.

5.1 Background

The *honeybits* open-source software provided in [6] is a simple tool developed in 2017 by a software developer named *Adel Karimi*, with alias *0x4d31*. The initial purpose behind the design of this tool, according to the developer in [6] was in verbatim to: *"initially 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 techniques studied [6].

The honeybit tool can be used for various tasks and purposes. However, it's 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. This cost-effective system is in response to find an affordable way to re-channel risk of compromise or intrusion on 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 [6]. The *honeybits* tool enables the defender to automate the design and creation of electronic decoys, which are hard to distinguish from real objects. In a way, these tokens behave just like honeypots, in the sense they not useful unless interacted with, and 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 form the tokens are, they are fake, bogus and of no real value. For instance, the token could be that of a *Canadian Social Insurance Number* (SIN), this number could be embedded inside a database system as a honeytoken, 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 seem 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 those based on 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; 6) Fake registry keys.

In this thesis, we utilize 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, by re-channeling it to a decoy see (*section 4.10*). 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; this makes them very attractive to an adversary.

5.2 Project Structure

The honeybits project and by extension the adversarial honeytokens project 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:

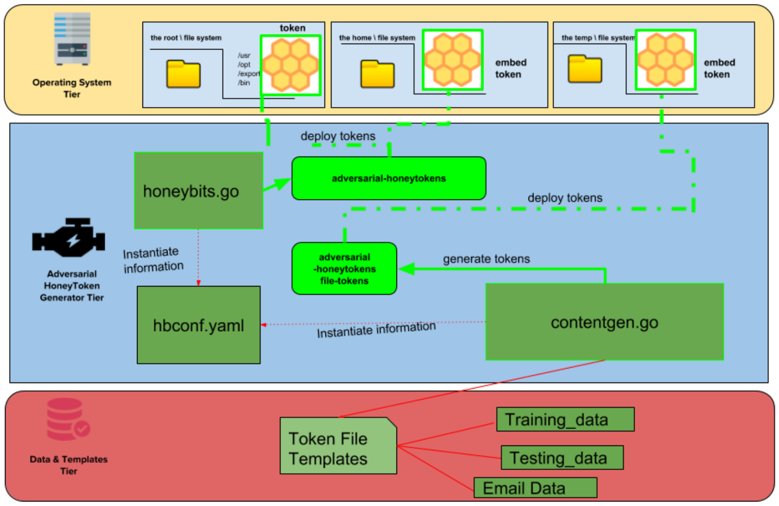


Figure 5.1: Adversarial HoneyToken Architecture

- contentgen/**contentgen.go -** this source code file is responsible for instantiating the IP address and 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.

- template /**testingdata -** is the sample template used to generate the testingdata file token to entice the attacker.

- **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 [6], and features and additions added by the thesis author (Fadi Younis):

- **Existing Features:**
  - i. Creating honeytokens that can be monitored.
  - 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 [6] 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 a bait token by the adversary, 3) sufficiently 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 adversarial examples transferability to 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 *host 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 include *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 honeytoken generator is its ability to inconspicuously implant artificial digital data (credentials, files, 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.

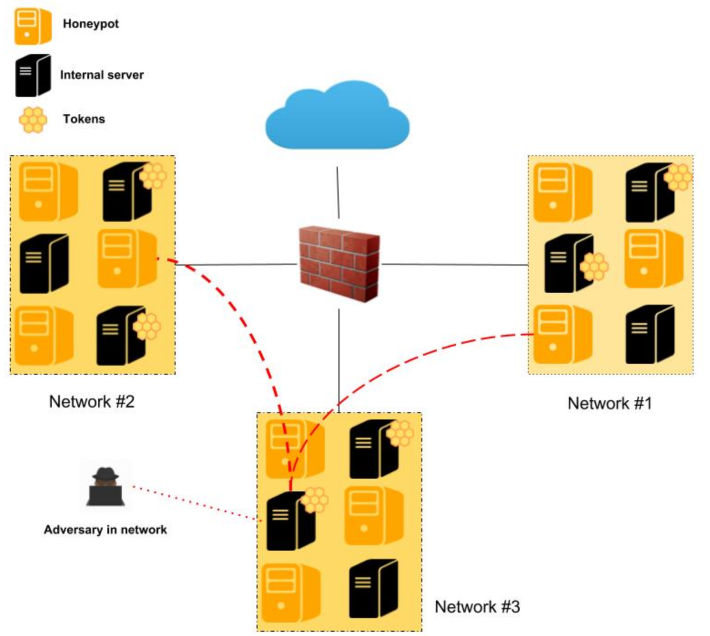


Figure 5.2: Adversarial Token Leakage

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.

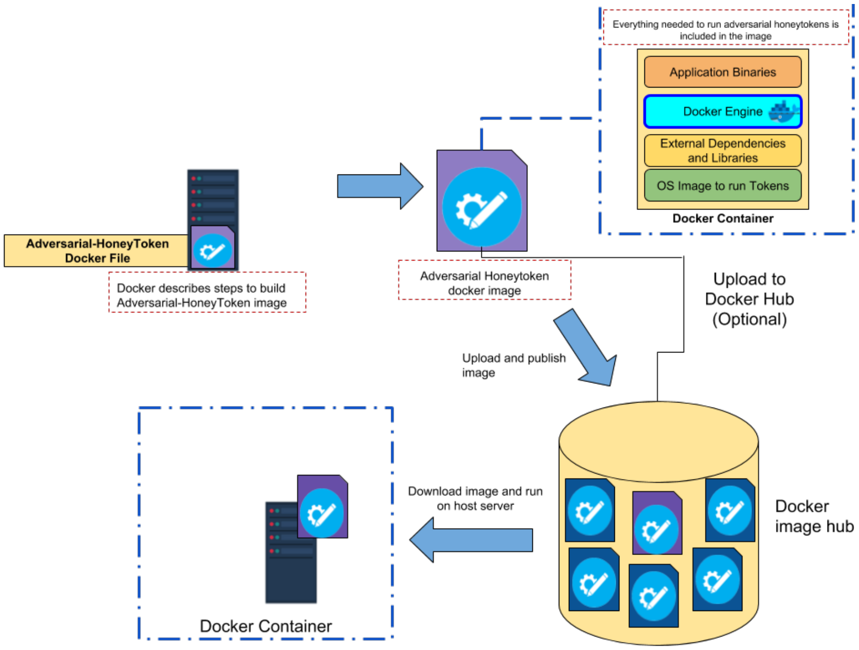

Everything needed to run adversarial honeytokens is included in the image
Application Binaries
Docker Engine
External Dependencies and Libraries
OS Image to run Tokens
Docker Container
Adversarial-HoneyToken Docker File
Docker describes steps to build Adversarial-HoneyToken image
Adversarial Honeytoken docker image
Upload to Docker Hub (Optional)
Upload and publish image
Download image and run on host server
Docker image hub
Docker Container

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. See Figure 5.3 above for an illustration of the Deodorized HoneyToken Environment. They will open via an exposed TCP/IP port open to the public, with weak non-privileged access points. The docker container will act as the *sandbox*, acting as entire layer to envelop the honeytoken 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 T_{decoy}. 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 the first 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 themselves, 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 below:

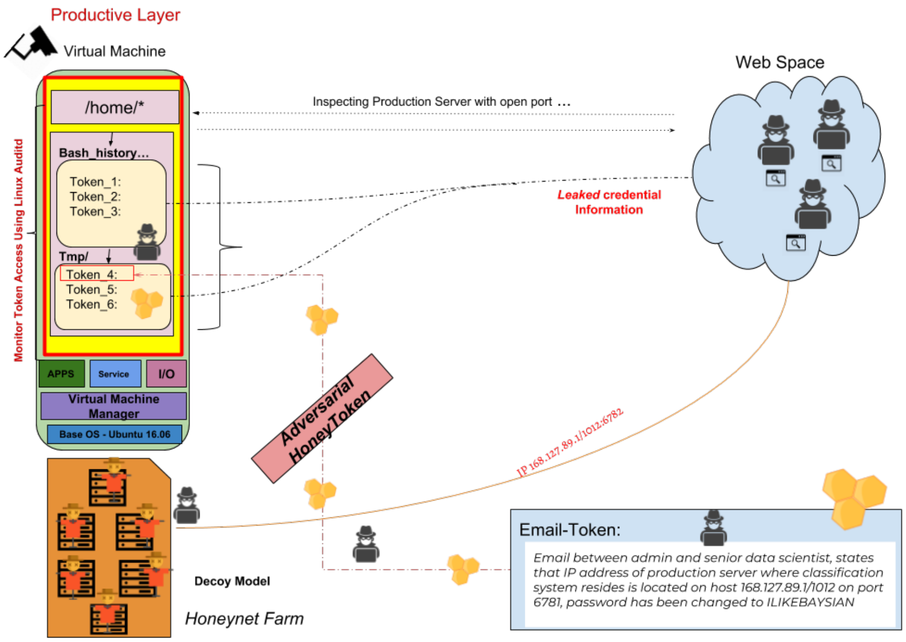
scenario 1 - Luring Away an Unwanted Adversary

- ***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 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.



Email between admin and senior data scientist, states that IP address of production server where classification system resides is located on host 168.127.89.1/1012 on port 6781, password has been changed to ILIKEBAYSIAN

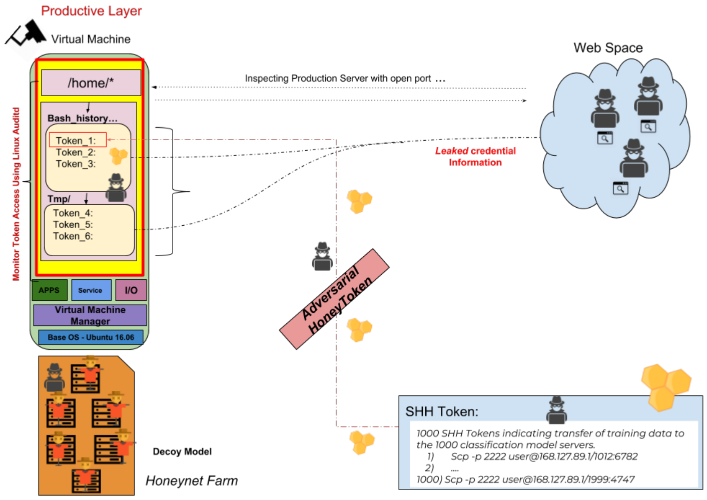
Figure 5.4: Scenario 1 - Luring Away Attacker from Target Model

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

- *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



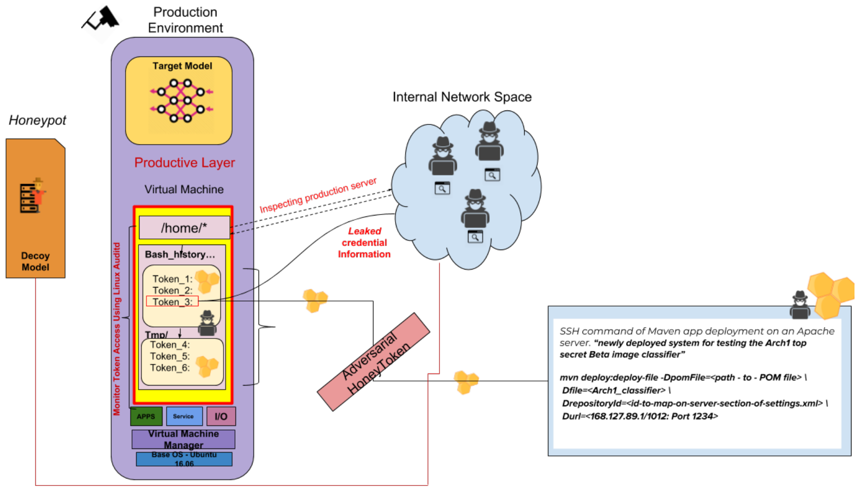
1000 SHH Tokens indicating transfer of training data to the 1000 classification model servers.
1) Scp -p 2222 user@168.127.89.1/1012:6782
2) ....
1000) Scp -p 2222 user@168.127.89.1/1999:4747

Figure 5.5: Scenario 2 - discourage future attacks

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 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.

5.7 External Dependencies

The adversarial honeytokens project has the following outside external dependencies:



SSH command of Maven app deployment on an Apache server. ***"newly deployed system for testing the Arch1 top secret Beta image classifier"***

```
mvn deploy:deploy-file -DpomFile=<path - to - POM file> \
 Dfile=<Arch1_classifier> \
 DrepositoryId=<id-to-map-on-server-section-of-settings.xml> \
 Durl=<168.127.89.1/1012: Port 1234>
```

Figure 5.6: Scenario 3 - apprehend an internal adversary

- **G0 Lang -** an open source programming language [32] 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 [1] 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 [4]. 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 [4]:
  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 [3]. 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 [2]:
  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 [5]. 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, as an added 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 debase 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 and discerning it from a decoy.

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 others 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 well-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 system. 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

Conclusions and Future Work

We have shown it to be possible is to use deception to prevent an adversary from mimicking a target model's classification behavior, if we successfully re-channel adversarial transferability. We have also presented a novel defense framework that essentially lures an adversary away from the target model, and blocks adversarial transferability, using various deception techniques. As discussed in *Chapter 4*, we can create an infeasible amount of computational work for the adversary, with no useful outcome or benefit to him. This can be accomplished by presenting the attacker with a hard *non-convex* optimization problem, similar to the one used for generating adversarial samples. Our framework allows the adversary to transfer these examples to a remote decoy learning model, deployed inside a high-interaction-honeypot. We believe the deception techniques in our framework are sufficient enough to fool adversaries, but we understand that other superior methods may exist. For the problems mentioned below it will be necessary to conduct direct research on adversarial defenses.

Firstly, in (*section 6.1*) we provide a conclusion to the work in our thesis. Then in (*section 6.2*), (*section 6.3*), and (*section 6.4*) we provide future research directions for our work.

6.1 Conclusion

In this thesis, we have discussed adversarial transferability of malicious examples, and proposed a defense framework to counter it, using deception derived from existing cyber-security techniques. Our approach is the first of its kind to use methods derived from cyber-security deception techniques to combat adversarial examples. We have also provided an implementation of one of the components in our framework (adversarial honeytokens). In this final chapter, we have highlighted some of the areas we believe to be important for the development of our framework and other defense techniques, such as such as the proposed use of signature trained classifier and using low-risk alternatives to honeypots.

We have it to be possible to develop an adversarial defense framework that poses as a secondary-level of prevention to curb adversarial examples from corrupting the classifier, used to deceive the attacker. we proposed a decentralized network of high-interaction honeypots as a decentralized defense framework that prevents an adversary from corrupting the learning model, primarily through the use of deception. We accomplish our aim by preventing the attacker from correctly learning the labels and approximating the architecture of the black-box system, luring the attacker away, towards a decoy model, using HoneyTokens, and creating infeasible computational work for the adversary.We hope what we propose in this chapter will help provide an interesting starting-point for future research in the field.

6.2 Signature-Based Classifiers for Adversarial Detection

As mentioned in *Chapter 4*, one of the greatest challenges in our thesis was deciding how to adequately detect, classify and label adversarial behavior as malicious. Simply building and training the actual classification model that would be responsible for doing so would have been a great undertaking on its own. However, there were other practical detection methods at our disposal, such as using signature-based detection to compare an object's behavior against a *blacklist*, and anomaly-based detection to compare an object against a *white-list*. We chose to lean towards the latter method (white-list) over blacklisting since we did not have reliable adversarial data that could have been used to generate a signature to fingerprint potential adversarial behavior.

Using a custom-built classifier trained with malicious behavior signature data could potentially provided a better alternative to *white-list* anomaly detection. The training-data set would not necessarily have needed to be complete, it could contain partial signatures, as long as it could help aid the classifier in detecting activity manifesting as malicious behavior, such as *injecting adversarial examples* or *querying the Oracle O*. Building and deploying this type of classifier can be extremely useful, as it can help detect new trends and variants in adversarial example attacks whose behavior might resemble or partially match the signatures learned by the classifier.

6.3 Reduced-Risk Alternatives to Honeypots

As we saw in *Chapter 4*, our framework utilizes *High-Interaction Honeypots* (HIHP) which simulate an actual full system for an adversary to interact with and exploit. These simulated systems include real OS, applications, input/output and services. However, simulating a live system is time-consuming and increasingly complex to build. A synthetic environment might make it sound like an attractive characteristic, but these types of honeypots utilize more resources than other honeypots, exhausting the production infrastructure. Deploying high-interaction honeypots imposes a high level of risk on the production system host environment when deployed. This becomes more of an issue if the adversary detects a trap and decides not to interact with the honeypot. But the more pressing matter is if the adversary decides to aggressively launch an attack against the entire production system in retaliation. It would be beneficial to find an alternative to high-interaction honeypots, which still maintain the same realistic environment, to entice the adversary. The greater issue is being able to mitigate the risk should a honeypot be overtaken by an advanced adversary. Perhaps the solution is to use a high and medium interaction hybrid, which would lower the level of risk since medium interaction honeypots only provide partial-access to the system.

6.4 Defending Against White-Box Attacks

Adversarial transferability is critical for black-box attacks, as adversaries attacking these systems are dependent on the high-rate of its success. In *Chapter 4*, we designed a framework to defend against an adversary with the ability to build a substitute training model with synthetic labels augmented from observing and collecting the labels of test samples from the *Oracle O*, despite the DNN model and training dataset being inaccessible. However, it is interesting to know how our deception framework would fare against an adversary with full-knowledge of the implementation of the learning model, except that it's deployed inside a honeypot. This means the adversary would have knowledge of the learning algorithm, training data, parameters, and full-access to the feature representation. Obviously adversarial transferability would become unnecessary to the adversary since he can build the examples directly on the target model. It would be interesting to see if it is possible to build a defense framework to thwart adversaries with perfect knowledge. There is no doubt that adversaries are becoming more advanced, and an adversary with perfect knowledge would not be out of the ordinary in the near future.

Appendix A

Appendices

A.1 Adversarial HoneyToken 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*.

A.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 stderror 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 confguration 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"), conf.GetString("contentgen.rdpconn.domain"),
   conf.GetString("contentgen.rdpconn.pass"))
//TEXT EMAIL CASE
case "txtemail":
//Get the template data from textemail 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"), conf.GetString("contentgen.txtemail.pass"))

//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"), conf.GetString("contentgen.testing.path"))
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.
   testing.date"), 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"), conf.GetString("contentgen.training.path"))
*p = fmt.Sprintf(template, addr, conf.GetString("contentgen.
   training.date"), conf.GetString("contentgen.testing.date"))
//DEFAULT CASE - HELLO WORLD
default:
p := &data
*p = "Hello World!"
}
//return the data
return data
}
```

A.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 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 reader of this email, and any attachments
   , is not the intended recipient , you are hereby notified
   that any review, dissemination or copying of this email and
   any attachments is prohibited. If you have received
   this email and any attachments in error, please notify the
   sender by email or telephone and delete it from your
   email client.
```

A.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), template file path (/tmp/sampletemplate.txt)
— /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 multimon:i:1\nsession bpp:i:24\nfull address:s:%s\
   ncompression:i:1\naudiomode:i:2\nusername:s:%s\ndomain:s:%s\
   nauthentication level:i:0\nclear password:s:%s\ndisable
   wallpaper:i:0\ndisable full window drag:i:0\ndisable menu anims
   :i:0\ndisable themes:i:0\nalternate shell:s:\nshell working
   directory:s:\nauthentication level:i:2\nconnect to console:i:0\
   ngatewayusagemethod:i:0\ndisable cursor setting:i:0\nallow font
    smoothing:i:1\nallow desktop composition:i:1\nredirectprinters
   :i:0\nprompt for credentials on client:i:1\nuse redirection
   server name:i:0"
# 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,2017at21:59:15GMT+11To:JOHNDOE<dave.cohen@example.com
    >Cc:security<security@example.com>Hi,Ah,
    sorryIforgottosendyouthenewaddress:http://%
    sIalsoresetyourpassword(user:%s)tothedefaultpass:%
    sPleasesettheMFA(multi−factorauthentication)ASAP.Cheers,
    AdelOnApril18th2017,at9:57pm,admin<dave.cohen@example.com>wrote
    :Hiadmin,IjustwantedtologintotheMonitoringsystem,
    butIget404error.Couldyoupleasehavealookatit?
    ThanksDaveTheinformationcontainedinthisemailandanyattachmentsisconfidentia
    /orprivileged.
    Thisemailandanyattachmentsareintendedtobereadonlybythepersonnamedabove
    .Ifthereaderofthisemail,andanyattachments,
    isnottheintendedrecipient,youareherebynotifiedthatanyreview,
    disseminationorcopyingofthisemailandanyattachmentsisprohibited.
    Ifyouhavereceivedthisemailandanyattachmentsinerror,
    pleasenotifythesenderbyemailortelephoneanddeleteitfromyouremailclient
    .

#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
sshpass: true                                                        #
   PASSWORD ENABLED
pass: 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 on port 2121
user: admin                                                         #
   USERNAME
pass: 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 2222
user: root                              # root user
remotepath: /path/to/source             # REMOTE PATH
localpath:  /path/to/destination        # DESTINATION PATH
sshpass: true                           # PASSWORD ENABLED
pass: 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 2222
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 port
- 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_images; tar −xf −)'
```

A.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.AutomaticEnv()

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, kvsdir, kvskey)
} else {
conf.AddRemoteProvider(kvsprovider, kvsaddr, kvsdir)
}
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 string) {

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 ”sshpass”:
sshserver := conf.GetString(”honeypot.addr”)
if p := &sshserver; conf.IsSet(”honeybits.ssh.server”) {
∗p = conf.GetString(”honeybits.ssh.server”)
}
honeybit := fmt.Sprintf(”sshpass −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 %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\naws %s −−profile %s −−region %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", "perm=wra", "-k", "honeyfile").Run()
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−audit for Mac OS\n”)
}

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”, fp)
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.confile")
bhpath      = conf.GetString("path.bashhistory")
hostspath   = 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.monitor: go-audit, auditd, none\n")
}
}
// [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", awsconfpath, 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)
}
}
}
```

Bibliography

[1] goenv: Isolated development environments for Go. `https://github.com/crsmithdev/goenv`. Accessed on March 13, 2018.

[2] Linux audit userspace repository. `https://github.com/linux-audit/audit-userspace`. Accessed on March 25, 2018.

[3] Store and retrieve encrypted configs from etcd or consul. `https://github.com/xordataexchange/crypt`. Accessed on March 13, 2018.

[4] viper: Go configuration with fangs. `https://github.com/spf13/viper`. Accessed on March 25, 2018.

[5] What is Docker? `https://www.docker.com/what-docker`. Accessed on March 3, 2018.

[6] Adel Karimi. honeybits. `https://github.com/0x4D31/honeybits`. Accessed on March 27, 2018.

[7] M. Akiyama, T. Yagi, T. Hariu, and Y. Kadobayashi. HoneyCirculator: Distributing credential honeytoken for introspection of web-based attack cycle. *International Journal of Information Security*, 17(2):135–151, Apr. 2018.

[8] R. M. Campbell, K. Padayachee, and T. Masombuka. A survey of honeypot research: Trends and opportunities. In *Proceedings of the 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 208–212, Dec. 2015.

[9] N. Carlini and D. Wagner. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 3–14, New York, NY, USA, 2017. ACM.

[10] N. Carlini and D. Wagner. Defensive Distillation is Not Robust to Adversarial Examples. In *Proceedings of the International Conference on Learning Representations (ICLR), 2017*, pages 1–3, 2017.

[11] S. Dowling, M. Schukat, and H. Melvin. A ZigBee honeypot to assess IoT cyberattack behaviour. In *Proceedgings of the 2017 28th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2017.

[12] A. A. Egupov, S. V. Zareshin, I. M. Yadikin, and D. S. Silnov. Development and implementation of a Honeypot-trap. In *Proceedings of the 2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 382–385, Feb. 2017.

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy. EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES. *International Conference on Learning Representations, 2017*, page 11, 2015.

[14] 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.

[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] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar. Adversarial Machine Learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, AISec '11, pages 43–58, New York, NY, USA, 2011. ACM.

[17] C. Irvene, D. Formby, S. Litchfield, and R. Beyah. HoneyBot: A Honeypot for Robotic Systems. *Proceedings of the IEEE*, 106(1):61–70, Jan. 2018.

[18] 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.

[19] A. Kurakin, I. J. Goodfellow, and S. Bengio. ADVERSARIAL EXAMPLES IN THE PHYSICAL WORLD. *International Conference on Learning Representations (ICLR), 2017*, page 14, 2017.

[20] M. A. Lihet and V. Dadarlat. How to build a honeypot System in the cloud. In *Proceedings of the 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, pages 190–194, Sept. 2015.

[21] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. In *Proceedings of the International Conference on Learning Representations, 2017*, page 14, 2017.

[22] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proceeding of the International Conference on Learning Representations (ICLR 2018)*, 2018.

[23] 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.

[24] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, June 2015.

[25] 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.

[26] 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.

[27] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical Black-Box Attacks Against Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 506–519, New York, NY, USA, 2017. ACM.

[28] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 372–387, Mar. 2016.

[29] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597, May 2016.

[30] S. Rauti and V. Leppnen. A survey on fake entities as a method to detect and monitor malicious activity. In *Proceedings of the 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 386–390, Mar. 2017.

[31] M. Ribeiro, K. Grolinger, and M. A. M. Capretz. MLaaS: Machine Learning as a Service. In *Proceedings of the 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 896–902, Dec. 2015.

[32] Robert Griesemer. The Go programming language. `https://github.com/golang/go`. Accessed on March 13, 2018.

[33] A. Rozsa, M. Gunther, and T. E. Boult. Towards Robust Deep Neural Networks with BANG. *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV), 2018*, Nov. 2016. arXiv: 1612.00138.

[34] 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.

[35] X. Suo, X. Han, and Y. Gao. Research on the application of honeypot technology in intrusion detection system. In *Proceedings of the 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*, pages 1030–1032, Sept. 2014.

[36] 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.

[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. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 39–49, New York, NY, USA, 2017. ACM.