Diverse, Neural Trojan Resilient Ecosystem of Neural Network IP

BROOKS OLNEY, University of South Florida, FL
ROBERT KARAM, University of South Florida, FL

Adversarial machine learning is a prominent research area aimed towards exposing and mitigating security vulnerabilities in AI/ML algorithms and their implementations. Data poisoning and neural Trojans enable an attacker to drastically change the behavior and performance of a Convolutional Neural Network (CNN) merely by altering some of the input data during training. Such attacks can be catastrophic in the field, e.g. for self-driving vehicles. In this paper, we propose deploying a CNN as an *ecosystem of variants*, rather than a singular model. The ecosystem is derived from the original trained model, and though every derived model is structurally different, they are all functionally equivalent to the original and each other. We propose two complementary techniques: *stochastic parameter mutation*, where the weights θ of the original are shifted by a small, random amount, and a *delta-update* procedure which functions by XOR'ing all of the parameters with an update file containing the $\Delta\theta$ values. This technique is effective against transferability of a neural Trojan to the greater ecosystem by amplifying the Trojan's malicious impact to easily detectable levels; thus, deploying a model as an ecosystem can render the ecosystem more resilient against a neural Trojan attack.

Additional Key Words and Phrases: Adversarial Machine Learning, Data poisoning, countermeasures

ACM Reference Format:
Brooks Olney and Robert Karam. 2021. Diverse, Neural Trojan Resilient Ecosystem of Neural Network IP. *ACM J. Emerg. Technol. Comput. Syst.* 1, 1, Article 1 (January 2021), 23 pages. https://doi.org/10.1145/3471189

1 INTRODUCTION

The field of Artificial Intelligence (AI) and sub-field of Machine Learning (ML) have been one of the most important and prolific areas of research in the past few decades. Within the field of ML, the area of deep learning, and specifically with Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) has seen great success in areas such as healthcare, security, speech recognition, and computer vision. Their applications in computer vision, namely, image recognition and classification, have seen some of the most widely recognized advancements in the area [60, 27, 21, 18]. Advancements in the automotive industry have also brought about the concept of self-driving cars, which rely heavily on neural networks for processing their surroundings and making decisions. The innovation within this industry has sparked interest in research geared towards equipping such autonomous systems with high performance and fault tolerant deep learning algorithms [8]. Operational failure of these systems can potentially lead to injury and property damage, and thus may introduce liability issues.

This is an important consideration with regards to security as well. Security concerns arise when an intellectual property (IP) vendor obtains an untrusted ML model and deploys it on their systems,

Authors' addresses: Brooks Olney, brooksolney@usf.edu, University of South Florida, Tampa, FL; Robert Karam, rkaram@usf.edu, University of South Florida, Tampa, FL.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1550-4832/2021/1-ART1 $15.00
https://doi.org/10.1145/3471189

with no knowledge of how it may have been tampered with. Concerns also arise from situations where an IP vendor finalizes their own models and deploys them on their systems, and they must operate within an untrusted environment. In response, Adversarial Machine Learning has been a highly influential topic of research within the ML community in recent years. Attacks on ML systems can be divided into two main categories based on the influence of the attack [24], namely: 1) *causative attacks* : which aim to disrupt the learning process of the network at training time, and 2): *evasion attacks* : which aim to exploit the classifier during inference. Poisoning is a causative attack that provides contaminated or mislabeled data points in the training set, degrading the accuracy of the classifier. Evasion attacks are an example of exploratory attacks that attempt to construct input samples which are designed to trick the classifier into interpreting them as something entirely different, or just to gain information about how the ML algorithm works.

While poisoning techniques generally evaluate their strength based on the error rate induced on the classifier, Neural Trojans [36] are a sub-category of poisoning techniques which embody similar characteristics to that of hardware Trojans. They are extremely difficult to detect because they are only activated by a certain set of inputs that are only known to the adversary, and the threats posed by such attacks can be catastrophic. For instance, a neural network IP may be used in an autonomous vehicle to classify street signs, and an embedded Trojan may cause the classifier to recognize a stop sign as a 75mph speed limit sign when a sticker is on the sign. However, one drawback of neural Trojan approaches is that they assume the attacker has access to the input data to the network. We believe that a neural Trojan that does not require direct manipulation of the input data for triggering is a much more difficult threat to detect.

In addition to the practicality of the attack itself, there may be thousands or even millions of copies of that model in operation in the wild. If an attacker is able to successfully mount an attack on one device, then they could also transfer this attack to any other identical device. Typically, *transferability* of an adversarial attack refers to the ability of an *adversarial example* to fool multiple different models with different architectures, or for a *poisoning* attack to be applicable to multiple different network architectures [13, 42, 34, 51, 38]. In the context of this paper, we use the term transferability in the same fashion, but with the assumption that if an attacker can mount a poisoning attack on a system, then they should be able to transfer that attack to an *identical* system with ease. This aspect of transferability is commonly overlooked in typical threat analyses. We believe it paramount to consider the *spread* of a devastating attack before a proper solution has been implemented - especially in the aforementioned autonomous systems.

In summary, the main contributions of this paper are as follows:

1) We define a more practical and devastating threat model for which neural network Trojans can be applied.
2) We present a novel method of deploying a CNN as an *ecosystem* which is derived from the original CNN through a process called *stochastic parameter mutation*.
3) We propose *delta-update*, a method of updating CNNs in the field which exposes less information about the classifier to an attacker, and effectively thwarts a poisoning or neural Trojan based attack on the model ecosystem.
4) We demonstrate that, with adequate self-check procedures and pooled validation across the ecosystem, a stealthy attack on one model cannot be translated to any other models within the diversified ecosystem while maintaining the stealthy nature of the attack. We validate this claim with statistical analysis of the resulting accuracy before and after the adversarial update has been applied to every other model in the ecosystem.

The rest of the paper is organized as follows: Section 2 provides a background on topics related to adversarial machine learning and security trends inspired by biology. Section 3 describes the

threat model, as well as who the attacker is and what their capabilities and goals are. In Section 4, we present the methodology of our approach. Then, in Section 5 we present the results of both the diversity and security of the ecosystems that were generated. In Section 6, we perform a security analysis of our approach. Finally, we elaborate on possible future works conclude in Section 7.

2 BACKGROUND & RELATED WORK

In this section, we provide a thorough background on adversarial machine learning, including the current state-of-the-art in neural network Trojan attack and defense methods. Next, we provide a background on biologically inspired hardware and software security methods from which we draw inspiration. Finally, we describe several related works.

2.1 Adversarial Machine Learning

Adversarial machine learning is a relatively new area of research aimed towards exposing and mitigating the security flaws in ML algorithms and their implementations. The field of adversarial machine learning is indeed vast and encompasses many different concepts stretching across an array of domains. Attacks within this domain typically fall within 2 different categories of influence [24], namely: 1) Causative attacks, and 2) Exploratory attacks. Causative attacks are aimed towards subverting the training process by inserting malicious examples into the training set. The malicious examples may be created to be near the decision boundary between two or more classes and/or be generated using gradient-based methods, thus maximizing the error rate [7, 55, 17, 39]. They may also include certain noise or signatures which serve as a backdoor for the attacker to reprogram the network and cause targeted misclassification when it encounters an instance with the signature of the backdoor [9, 25, 32, 2, 19]. Such backdooring works are very closely related (perhaps even synonymous) to more recent neural network Trojan attacks [36, 35, 31]. Exploratory attacks are generally aimed towards exposing certain information about the underlying architecture of the learning model by providing adversarial examples [1, 58]. One type of exploratory attack is called an evasion attack. In an evasion attack, the adversary carefully constructs adversarial examples that are specifically made to evade or "fool" the learning algorithm [6, 49, 17]. Evasion attacks are perhaps the most thoroughly researched method of attacking a classification system, because the attacker does not need to make any changes to the underlying network structure or parameters, and adversarial examples may be transferable to similar model architectures. In this paper, we focus on subverting the *spread* of causative attacks, and specifically, neural Trojan-like attacks. A more in-depth explanation of the threat model, including the goals and capabilities of the attacker, is provided in Section 3.

2.1.1 Neural Trojans. In general, the process of backdooring or Trojaning a neural network involves training/retraining the network with a contaminated dataset. The training dataset will contain samples that have been embedded with the Trojan trigger, where the Trojan trigger is "noise" that is drawn from a different distribution than the uncontaminated training data. This trigger, when added to a sample image, is indistinguishable to the human eye. However, the model is inherently programmed through the training process to recognize this pattern and classify the sample as whatever the attacker wishes it to. Most importantly, the integration of the Trojan should have a *minimal* effect on the classification accuracy on normal testing data. Because the network performs completely to spec, and the trigger does not activate the Trojan when testing on drawn from the same distribution as the original training data, this makes the Trojan very difficult to detect.

Intuitively, neural Trojans share some of the same properties as hardware Trojans [50, 35]. Specifically, for most inputs, the neural network IP performs to specification, and the activation of the Trojan drastically alters the functionality of the neural network IP. However, one important

distinction between them is that, a hardware Trojan may be designed to trigger upon a particular (sequence of) input(s) that may occur naturally, without requiring the attacker to manually supply the trigger input (sequence). For a neural Trojan, the input images must contain the Trojan trigger for it to activate, which is an abnormal occurrence because the trigger is a signature that comes from a different distribution than anything it has seen before. This means that, the attacker must not only have access to (or replicate) the training data, they must also have the ability to influence the input data during runtime – adding an additional layer of complexity to the attacker.

There are several works that have been or can be categorized as Trojan attacks on neural networks. For example, the authors in [19] demonstrated that they could backdoor a neural network IP simply by adding some malicious examples to the dataset containing a pattern of bright pixels - without compromising the accuracy of the model. The authors also demonstrated that they could cause classifiers to incorrectly label stop signs that contain a sticker with the backdoor on the bottom of the sign. Liu et. al. [36] presented a Trojaning attack procedure that selects specific neurons that will be used to trigger the Trojan, and carefully crafts a Trojan trigger in such a way that maximizes the output values of the selected neurons. They demonstrated through this process that they could successfully Trojan a network even without access to the original training data or process. In [33], the authors proposed a bit-level attack called SIN^2 which involves embedding a software Trojan within the redundant space in the binary representation of the network. The Trojan can then be extracted from the network and executed on the target system to perform some malicious action. For example, the authors demonstrated that they could embed a "fork bomb" in the neural network and execute it on the target system. Similarly, the authors in [43] present an algorithm which generates a trigger that is specifically constructed to find "vulnerable" bits of the weights. They then perform bit-flip attacks (e.g. row-hammer) to those bits, causing the network to classify all inputs to a target class. They demonstrated that they could force a network to incorrectly classify 92% of test images to a target class with only 84 bit-flips out of 88 million total bits on ResNet-18.

2.1.2 Label Flip Attacks. Label flip attacks (LFA) were first published by Xiao et. al. in 2012 [55] in the context of poisoning attacks against SVM. They were able to optimize the label flipping strategy such that the number of contaminated samples is minimized, while the classification error rate is maximized. With a naive approach, such an attack is relatively easy for an attacker to perform, because it simply requires that labels from the training set be flipped. The result causes the classifier to mislearn certain features across the mislabeled classes. Further, the algorithm in [55] is optimized to impact the accuracy of the target model as much as possible with a minimal number of flipped samples. In this paper, we utilize a naive implementation of the label flip attack as a baseline in order to simulate the increase of error rate and change in network parameters after retraining due to its ease of implementation and architecture agnostic nature.

The LFA differs from traditional Trojan and backdoor approaches in the sense that *it has no explicit trigger*. Depending on the parameters for the LFA – i.e. the targeted labels, *which* samples are flipped, number of samples flipped, etc. – the attack can pose many different types of threats. For example, the attack can have a high impact on accuracy when a large number of samples are flipped. Or, the attack can be more subtle by flipping less samples but only if those samples lie close to the decision boundary for the targeted class. That being said, we believe that, because the LFA does not rely on a trigger to activate, that it can be viewed as an *always on Trojan*, which has the potential to be even more devastating if not discovered in time.

2.2 Bio-Diversity inspired Security

As computing systems become a more pervasive part of every day life, deploying these systems in a more secure manner becomes more and more difficult. This has inspired researchers to develop

newer and more sophisticated security protocols, including protocols that are inspired by biology and species diversity. There have been many research endeavors that propose bio-inspired security protocols for the secure deployment of hardware [26, 37, 59, 40] and software [11, 4, 28], as well as security protocols to protect networks and cyberspace [44, 15] and to ensure data privacy and security [22, 12]. In general, diversity of a species at the genetic level is necessary for the overall survival of that species. If a pathogen enters an ecosystem and there is no immunity, then the pathogen will spread and infect more and more organisms in the ecosystem, causing an epidemic. This can be extended to the realm of computer system security by recognizing the fact that attacks on one system can be transferable to another – similar or identical – system. Within the domains of internet of things (IoT), self-driving cars and others, it is important to consider that the hardware within these devices must remain in operation in the field for a long period of time, and that the underlying hardware in these products often consists of similar components. For example, given a classification model for a self driving car, there may be several hundred thousand or even millions of these products in operation, with each one having a similar or identical hardware composition. This has motivated researchers to investigate alternative means of hardware deployment, including mutating or "patchable" hardware [26, 40].

2.3 Related Work

There are several works which propose methods to detect and mitigate neural Trojans. For example, the authors in [36] describe several approaches, including; 1) *Input anomaly detection*, where SVMs and decision trees are used to discover if there are training samples that come from a different distribution than the rest of the training data. 2) *Re-training*, where the user retrains their network further with legitimate training data. and 3) *Input preprocessing*, where an autoencoder is used to filter the input data. In general, the methods to mitigate neural Trojan insertion and/or activation can be loosely categorized into these 3 main categories. Gao et. al. proposed STRong Intentional Perturbation (STRIP) to detect Trojan inputs during runtime [16]. They detect input anomalies by applying perturbations to the input image and measuring the entropy in classification for that input. Intuitively, a high amount of classification entropy indicates that the input is benign, while a low amount of entropy may indicate that the input contains a Trojan trigger. The authors in [3] propose a strategy similar to input preprocessing, but to the training data prior to training. They use "provenance data" which is used to group the training samples by the probability of them being benign or containing a Trojan trigger. Each group is then evaluated by training with and without it, to see what the affect on network accuracy is. This effectively removes the Trojan from the training set, and thus prevents it from being inserted into the network. Finally, the authors in [54] presented a framework of detecting Trojans in CNNs when access to the underlying training/testing data is either limited or nonexistant. Their approach is able to locate and thus reverse engineer the trigger by maximizing the affected neurons' outputs – similarly to our approach, where affected neurons may exhibit more significant errors than others.

There are also several works specifically targeting *backdoor attacks*, which are very similar to neural Trojans. One of the first examples, Neural Cleanse [53], presented a comprehensive framework for detecting, identifying, and mitigating backdoor attacks. They demonstrated that they could detect a backdoor by searching for the minimum trigger required to make all samples go to a target class. This proves to be quite effective, but can become computationally expensive with datasets that have a large number of labels. The authors in [61] proposed a method of detecting several different types of attack by using *mode connectivity*. They demonstrated that their approach was better than a method based on random weight perturbations, which is similar to our approach, however, their process of noise generation differs from ours, and we include additional measures beyond weight perturbations.

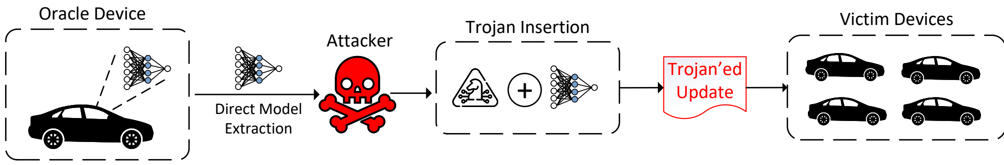


Fig. 1. Overview of the threat model. The attacker gains access to an "oracle" device which they can extract the classification model from. They can then mount their Trojaning attack on their extracted copy and generate a malicious update for the oracle. If the malicious update achieves the attacker's goal, then they proceed by uploading that update to every other identical device they are able to discover over a network.

All of these methods propose different solutions to the same question: "*how can we mitigate neural Trojans?*". However, they address this problem within the context of a single device or system that is being attacked, without considering the existence of thousands or even millions of identical devices in the wild. In this paper, we pose the question: "*how can we prevent the **spread** of a neural Trojan?*". To address this unmet need, we leverage the ambiguous and complex nature of neural networks to create a diverse ecosystem of "genetically" different but functionally equivalent networks, where a single network may be infected, but that infection will not spread to the rest of the ecosystem. To the best of our knowledge, there has been no prior work in literature that proposes a method of diversity for securely deploying neural networks, nor have there been efforts towards leveraging the ambiguity of their knowledge representation through the weights for security purposes. As such, the methods outlined in this paper serve as a novel contribution to this field. DiMattina et. al. first explored the concept of making small adjustments to the model while retaining its functionality in [14]. The authors proposed a biologically-inspired mathematical model to determine if a neural network can retain its functionality while undergoing gradual perturbation of its parameters and structure. Their results demonstrate that this is possible for networks where the neurons within hidden layers have gain functions that are power, exponential or logarithmic. In [41] the authors propose the use of genetic algorithms to search for an accurate and diverse set of learning models for ensemble learning. We leverage these concepts of gradual perturbation (mutation) of a network's parameters for the purpose of neural network deployment security, especially for ecosystem-wide neural Trojan resilience, and quantify the level of difference and functional equivalence due to these mutations.

3 THREAT MODEL

In this section, we provide an in-depth explanation about the assumptions we make about the system and the attacker. We begin by describing the system that the attacker intends to compromise, and then go into detail about what goals the attacker has. Finally, we will briefly enumerate over the capabilities of the attacker.

3.1 Target System

In general, the target system is a commercial product which contains a neural network for inference as a critical component of the overall functionality of the system. A practical example of the target system is a self-driving car, where the neural network inference engine is critical for the vehicle's ability to see and interpret street signs, markers, pedestrians, other cars, etc. In this scenario it is critical that the system be as fault tolerant as possible – misclassification during a key moment in time may have catastrophic consequences.

The target system should have some baseline security protocols in place to prevent against common means of tampering. For example, when a remote update of the system is in progress, the system may evaluate the neural network before and after applying the update on several different metrics related to performance, i.e. a built-in self test (BIST). Example metrics could be; loss and accuracy on a validation set, statistical comparison of the weight histograms, or even a checksum of the update file [5]. Presence of these basic security protocols is assumed as a baseline for deployment of the system.

3.2 Attacker's Goals

The attacker's primary goal is to implement a malicious update that bypasses all security protocols, and spread that update to as many other devices as possible. In general, this update can be referred to as the *neural Trojan*. The malicious intent of the Trojan is to induce targeted misclassification at a very low rate such that it is extremely hard to detect by the BIST. An example would be an attacker that has their own self-driving car and has learned how to breach its security mechanisms and send Trojan'ed updates to other users that own the same model of vehicle.

3.3 Attacker's Capabilities & Knowledge

As mentioned previously, we assume the attacker has access to the system. With this access, the attacker is able to apply inputs to the network and observe the outputs to the network. This is referred to as a *black-box attack*, where the attacker has limited to no knowledge about the architecture of the system, but can observe inputs and outputs. Through trial and error, the attacker can extract their own copy of the classification model by observing inputs and their corresponding outputs - which has been shown to be a practical assumption in recent literature [23, 52, 45, 56–58].

To mount the neural Trojan that we have proposed, the attacker must also have access to a subset of the training data, or at least some data for the given application that follows the same distribution as the original training data. Recent research efforts have shown that it is possible to extract a subset of the training data from a model given black-box access [48]. So, such an assumption is realistic. Using the data, the attacker will mislabel some subset of those examples corresponding to the target class labels, and retrain the network. The attacker can then generate and apply the update to the system - given that it is able to circumvent the baseline security protocols. An overview of the threat model is given in Figure 1.

4 METHODOLOGY

In this section, we describe our proposed methodology, beginning with a high level overview of our approach. We then explain how the mutation process works to generate "versions" or "variants" of the same network that are quite different structurally but remain nearly identical functionally. Finally, we describe the proposed mechanism for securely updating said networks in the field.

4.1 Overall Flow

We propose a novel framework designed to enable secure deployment of sensitive NN-IP. An overall flow of the approach is shown in Figure 2. This is enabled via two key components: namely, 1) *stochastic parameter mutation* (SPM), and 2) *delta-update*. First, SPM allows us to generate N *variants* of the same network by gradually perturbing its weights. This is performed repeatedly N times, until the entire ecosystem of models has been generated. The parameters of the ecosystem are then stored in a secure database, shown in Figure 2(a). In the next section, we will describe the SPM process in detail and *why* it does not affect the network performance significantly.

Next, the ecosystem is deployed to the appropriate devices, where each device has its own unique copy of the original NN-IP. During the lifetime of the device, improvements may be made to the

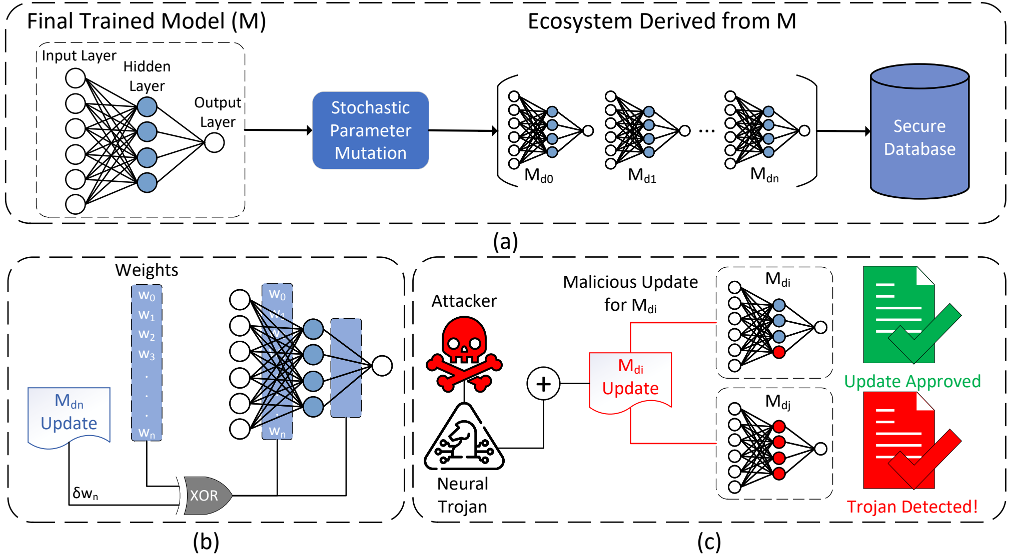


Fig. 2. (a) Diagram of the overall flow of the proposed methodology. Final network M is trained, then *stochastic parameter mutation* is applied to the weights to derive n copies of the model, which are stored in a secure database for future deployment. (b) Diagram of the xor-update procedure. Values for δw_n are XOR'd with the target network's weights. (c) Diagram showing that an attacker has determined how to insert a Trojan into model M_{dn}'s update file. The attacker then sends the malicious update to models M_{dn} and M_{dm}. Because the update was tailored for M_{dn}, it goes undetected by the built-in self test. However, when applied to M_{dm}, the update loses its stealthiness, and the test is able to determine that a Trojan has been inserted.

underlying NN-IP and thus the copy on each device would need to be updated to include these enhancements. To *securely* update these devices in the field, the second component of our approach, *delta-update*, ensures a 1-to-1 mapping between the device and its update file by XOR'ing the contents of the update file with the current network weights, shown in Figure 2(b). This 1-to-1 mapping ensures that a malicious update carefully crafted by an attacker *cannot infect every other network in the ecosystem*. For example, in Figure 2(c), we see that an attacker has Trojan'ed M_{di}, but that same update raises many flags when applied to M_{dj} because of the inherent sabotage caused by the XOR mismatch. We can leverage this to provide pooled validation across the entire ecosystem which gives the IP owner a clear indication that there is an issue with the given update.

Together, these algorithms implement our diversity-inspired method of secure deployment for NN-IP. In what follows, we will describe each process in detail.

4.2 Stochastic Parameter Mutation

Naturally, CNNs have evolved to a point where they require a very large number of trainable parameters (i.e. weights, activations, bias terms, etc.), for example, AlexNet [29] has a total of 62,378,344 trainable parameters. Assuming that the network parameters are in the form of 32-bit floating point, we can estimate the total size of an AlexNet model to be around 237 MB. The representation of the network in binary is inherently ambiguous, due to the structure and organization of the weights due to network architecture, as well their binary representation. Because of the nature of floating point numbers and how they are structured, small changes to a value may

result in an "avalanche effect" similar to that in the field of Cryptography, where, a small change in the decimal value will cause a large change in the binary representation of that number. In this paper, we leverage this behavior of floating point numbers and ambiguity of the model's weights to create a diverse ecosystem of classification models through the novel process of *stochastic parameter mutation*. With a CNN model M trained on $D_\tau : \{x_i, y_i\}_{i=1}^{k}$, and a user-defined shifting percentage threshold t_p, the process is formalized as follows:

Algorithm 1 Algorithm for Diversifying Model M

Input: M, t_p
Output: M_d
1: **for** layer in M **do**
2: **for** weight in layer **do**
3: $w \leftarrow weight$
4: $\delta_r \leftarrow w \times t_p$
5: $\Delta w \leftarrow$ Random uniform distribution $[\delta_r]$
6: Shift weight by adding Δw to w
7: **end for**
8: **end for**
9: **return** M_d

The histogram of the weights provides a nice visualization of their distribution, and thus the model itself. The distribution of weights in a neural network is determined by many different aspects of the given application such as; the architecture of the classifier, the training data, and the regularization types/parameters. That being said, the weights of the network typically follow a gaussian distribution with very small decimal values (e.g. $w_i = [-0.015, 0.015]$ for AlexNet [20]). This process alters the distribution of the weights in a nonlinear fashion. Figure 3(a) shows an example weight histogram before and after stochastic parameter mutation, and Figure 3(b) shows the histogram of the values for Δw.

Shifting the weights must be done on a per-weight basis, with a range specified by t_p. Applying this shift range to all of the weights at once may result in some weights being shifted by a greater value than intended - negatively affecting the accuracy. A random number generator is utilized for the computation of the shift amount Δw to ensure that every model M_d generated from M is unique. With the user defined value t_p, this shift amount is computed as a random value that is within the range of 0 and $w \times t_p$, meaning that if the user supplies a value of 0.01 (1%), then every weight in M will be shifted by *up to* 1% of that weight's value. Because the shifting is done randomly, there is a non-linear relationship in the distribution of weights from M to M_d. A pictorial representation of this process is provided in Figure 4.

Despite the shifting of network parameters, the network retains a nearly identical accuracy on testing data to the original model, with a relatively high average HD between M and M_d, which is shown in Section 5. It is important to note that the range of values for Δw is between δr and 0, rather than +/- δr. We decided on this range because for instances where the weights are shifted by a +/- amount, there is a much more pronounced effect on the accuracy of the resulting model. We believe that this is due to the polarity of weights on the inputs to a given neuron. If negative weights become more positive and vice versa, then the shift in the decision boundary for the neuron is much larger. In addition to this range, we decided to only apply stochastic parameter mutation to the weights, and not the bias terms. Avoiding the bias terms helps stabilize the outputs of each neuron, reducing the impact on accuracy. The results for poisoning trials are nearly identical, but omitting the bias terms from the process also has a positive impact on the baseline accuracy.

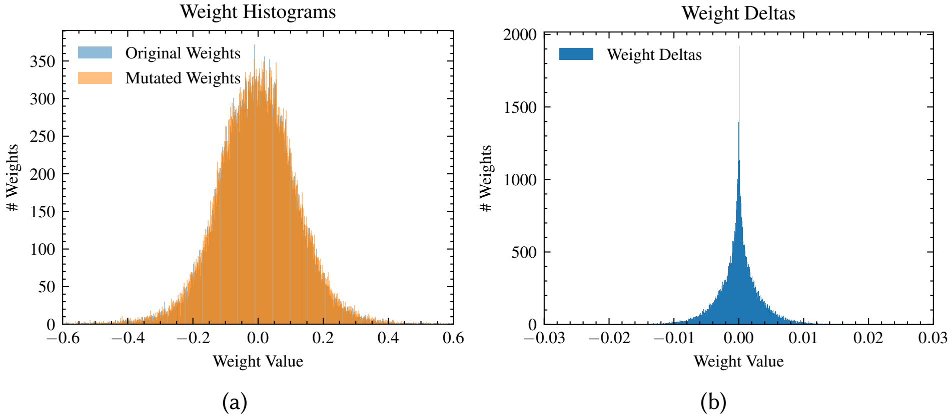


Fig. 3. Histograms of the weight distribution of a small MNIST classifier. Graph (a) shows the weights before and after stochastic parameter mutation is applied. Graph (b) shows the histogram of the Δw values. These results were generated using a value of $t_p = 0.10$ (10% diversity threshold).

Once the ecosystem has been derived from M, the weights for each network must be stored in a secure database; the flow for this part of the approach is shown in Figure 2(a). In the worst case, this incurs a total storage requirement of $n *$ (size of M). However, this may be alleviated by some data compression techniques which are outside the scope of this paper. It is an absolute necessity that the parameters for each model in the ecosystem be stored securely, as this will allow the manufacturer to issue personalized, secure updates to them in the field.

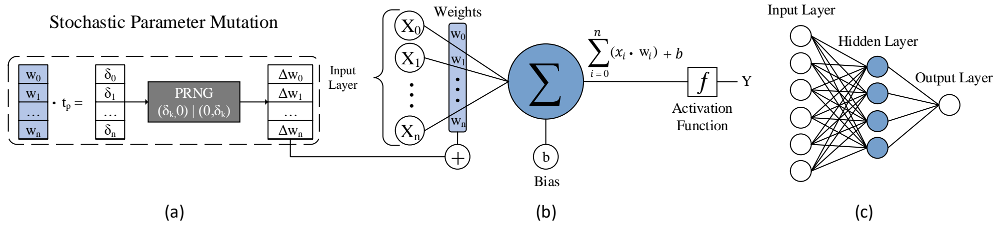


Fig. 4. (a) Diagram of *stochastic parameter mutation* applied to the neuron weights. The weights are first multiplied by the user defined threshold t_p to obtain $\{\delta_0, \delta_1, ..., \delta_n\}$ which is then passed into a pseudo-random number generator (PRNG) to supply the values for Δw_k. The PRNG produces values within the ranges of $(\delta_k, 0)$ and $(0, \delta_k)$ for values of δ_k that are negative or positive, respectively. The resulting values are then added to the original weights - effectively mutating them. (b) Standard diagram of a neuron, with the mutation process applied on the neuron weights. Output is computed by summing the dot product of the input layer with the weights and applying an activation function. (c) Example 1-layer feedforward network, with hidden neurons highlighted in blue.

4.3 Secure Update Mechanism

Traditionally, neural network models are updated by simply replacing the binary file of the network itself. The weights may be stored in a file and loaded on demand only if the network's structure has

not changed between initial deployment and the future update. Transmission of this network over the internet unintentionally exposes it to threats from a network-based attack, where an attacker may intercept the model, and maliciously modify it before continuing the transmission to the target device. Because of this, we believe that in some scenarios this update procedure may be infeasible due to security and privacy restrictions. More importantly, simply overwriting network parameters can enable an attacker to place their Trojan'ed network into another device. Hence, limiting the *transferability* of the Trojan can help ensure the security of the system.

To address these vulnerabilities, we propose a novel update mechanism which functions by transmitting the *deltas* of the weights, rather than the entire weights themselves to the target device containing the network. Instead of replacing all of the weights by simply loading the binary file, the binary file will be loaded into memory, and the weights will be updated by XOR'ing each weight with its corresponding value in the update file – as shown in Figure 2(b). The process of generating an update file for a network is as follows: first, the model will be retrained with new data, or different/new parameters that have been chosen to improve the performance of the network; next, the update file is generated by taking the matrices of weights before and after retraining and XOR'ing them together. This accomplishes two things, namely:

1) The amount of data potentially exposed to an attacker monitoring the network is minimized, with no way for the attacker to infer what the actual weights are.
2) The ability to update a neural network's weights is preserved, due to the XOR functionality.

Use of this update mechanism does not preclude additional precautions / layers of security, such as encrypting the update file prior to transmission or the use of digital signatures. *SPM, in conjunction with the proposed XOR-based update mechanism, ensures a 1-to-1 relationship between a model and its update file.* Because of the non-linearity associated with randomly shifting the weights, an update for model M_{di} applied to model M_{dj}, will result in unintended functionality to M_{dj} either through a degradation in accuracy, or even sabotaging the functionality of the model entirely. Most importantly, this scales to a practical threat scenario where an attacker attempts to Trojan multiple networks.

In the event that optimizations/improvements have been made to the accuracy of the original model, all of the models in the ecosystem should be updated to share these improvements. It is assumed that these devices would already have a built-in update mechanism. Rather than overwrite the existing parameters, the proposed technique would XOR a unique,targeted update file with the existing network parameters. Hence, for software-driven networks, or software-driven but hardware accelerated networks, the occasional update process may incur a slight delay due to the XORing of parameters, rather than simply overwriting, but this would not lead to any additional delay or overhead during inference.

In order to form the unique update files, the manufacturer must first taking the new version of M and apply SPM to derive the new ecosystem $M_{ec} = \{M_{d0}, M_{d1}, ..., M_{dn}\}$. Next, each model from the "new" ecosystem is mapped to each model in the "old" ecosystem by XOR'ing their parameters. This will derive an update for each model that can be deployed to the ecosystem. Finally, the existing ecosystem database must be overwritten with the new parameters. This will ensure consistency moving forward for future updates.

In addition to the XOR-based update functionality, an additional mechanism for evaluating the change in accuracy and/or training parameters should be in place to safeguard against a potentially malicious update. It is expected that many devices which require high reliability will retain some form of built-in self test (BIST) and conduct the test on startup. No valid update should have a negative impact on the accuracy of the model. So, the device running the network should attempt to update the model via the XOR-based method, and test the accuracy before and after the update.

If the accuracy falls by a certain amount (or even at all), then that update should be considered at least unfit, or potentially malicious, and discarded.

5 RESULTS

In this section, we describe our experimental setup, including specifications for the classifier used, as well as the method and parameters of the poisoning attack. We also present our results for the 2 separate tests, namely 1) show the effect of mutation on the performance (accuracy) and diversity (Hamming distance) of the ecosystem; and 2) examine the transferability of a Trojan by applying a malicious update intended for one model to another model. These tests effectively demonstrate that our approach ensures a sufficient level of diversity, and resilience to infection from another model's update without compromising functionality of the system.

5.1 Experimental Setup

As a baseline, we built a CNN based on the classic LeNet-5 model [30] for MNIST digit classification. The full structure of the network and total number of parameters are shown in Table 1. The model consists of two 2D convolutional layers with maxpooling layers after them, two fully-connected layers, and a softmax layer. On average, this simple model achieves 99% accuracy on the test set with a relatively short amount of training time.

In comparison to state-of-the-art computer vision model architectures, this example is quite small, and thus requires very little computational power. However, because more standard computer vision tasks in industry require more computationally expensive models, there has been a lot of research in network quantization for processing these workloads at the edge. Quantization of neural networks is aimed towards minimizing storage requirements and computation time by reducing the precision of the network. Hence, we apply our approach to the the MNIST classifier quantized to single-precision (32-bit) and half-precision (16-bit) floating point representation to demonstrate the scalability of our approach.

In addition, we have tested the proposed technique using the much larger VGG-16 based classifier fine-tuned on the CIFAR-10 dataset[1]. The full structure of this model and total parameters are shown in Table 2. This model has just over 15M total parameters, and was able to achieve a baseline performance of 96.53% on the testing set.

All of the software implementation for our experiments was done using Keras [10] – a high level library for interfacing with Tensorflow. Processing of these workloads was done using the Gaivi computer vision cluster containing GTX 1080 Tis and Titan V/Xs, at University of South Florida.

Table 1. Architecture of MNIST Classifier [30].

Layer type	Activation*	Neurons	Trainable Parameters
Conv	relu	6	60
Max-pooling	-	-	-
Conv	relu	16	880
Max-pooling	-	-	-
FC	relu	120	48,120
FC	relu	84	10,164
FC	softmax	10	850
Total	-	-	60,074

*Note: Original LeNet architecture uses Tanh activations instead of ReLU.

Table 2. Architecture of VGG-16 + CIFAR-10 Classifier [47].

Layer* type	**Activation***	**Neurons**	**Trainable Parameters**
Conv	relu	64	1792
Conv	relu	64	36928
Max-pooling	-	-	-
Conv	relu	128	73856
Conv	relu	128	147584
Max-pooling	-	-	-
Conv	relu	256	295168
Conv	relu	256	590080
Conv	relu	256	590080
Max-pooling	-	-	-
Conv	relu	512	1180160
Conv	relu	512	2359808
Conv	relu	512	2359808
Conv	relu	512	2359808
Conv	relu	512	2359808
Conv	relu	512	2359808
FC	relu	512	262656
FC	softmax	10	5130
Total	-	-	15,001,418

*Note: Batch normalization in between conv layers, before activations.

5.1.1 Hyperparameter Configurations. As mentioned in Section 2, we simulated a label flipping attack (LFA) which targets samples of targeted labels performed during the retraining phase of the network. Per [46], the batch size can *smooth* the results of a label flipping attack. With a higher batch size, the number of contaminated samples per batch will be lower, meaning that the resulting loss computed during back-propagation will be lower, and thus the error from the contaminated samples will be lower. So, it may be possible to achieve the attacker's goals using several different hyperparameter strategies which have different effects on the weights during poisoning.

Because we reuse the original training data for the LFA, there is no deviation in the distribution of the data used for training and retraining. As a result, the weights in the network may see minimal change while performing the attack. While we assume the attacker may try to optimize for stealthiness, we perform our experiments with multiple configurations of training hyperparameters in order to demonstrate the efficacy of our approach with different weight update strategies. Those configurations are shown in Table 3. For 2 of the configurations, we used a standard learning rate of $1e-3$, and for the other 2 configurations we chose a higher learning rate of $5e-3$ with a shorter training time in order to speed up convergence. The trials with a higher learning rate will of course result in larger initial updates to the weights, and so the difference in accuracy when applying the poisoned update may be larger. The results for these tests are shown later in this section.

[1]Model weights were obtained from https://github.com/geifmany/cifar-vgg.

Table 3. Hyperparameter configurations

Batch size	Epochs	Learning Rate
128	50	1e-3
128	25	5e-3
1024	100	1e-3
1024	50	5e-3

5.2 Diverse Networks

Using the methods that were described in Section 4.2 and the baseline MNIST classifier, we ran simulations on a range of values $t_p = [1.0, 10.0]$ in increments of 0.2%. For each value of t_p, we generated an ecosystem containing 1000 mutated versions of the original model and compared the accuracy against the original. Given these tests, we are primarily concerned with 2 metrics:

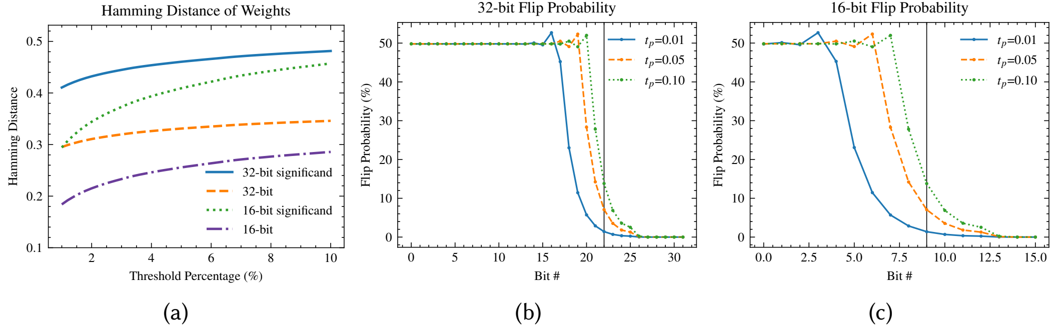
1) The HD between the weights of models A and B should be as close to 50% as possible.
2) The difference in test accuracy between models A and B should be as close to 0% as possible.

HD is an appropriate metric for measuring the similarity/dissimilarity between two strings. In this case, the more diverse and unpredictable the specific weights are for a given network *relative* to another in the ecosystem, the less likely an attacker would be able to leverage information from one known network to design an update file that is appropriate / targeted to another. In what proceeds, we evaluate our approach against these metrics.

5.2.1 Ecosystem Diversity. The average HD can be used as a metric to evaluate the ambiguity between models from the perspective of an attacker, and should ideally be 50%. In Figure 5(a) we show the average HD for the ecosystem at both precisions, and in Figure 5(b-c) we show the probability for each bit of the weight value to flip during the mutation process. In these graphs, the vertical line is positioned at the point on the x-axis corresponding to the final value bit of the given floating point precision, using the IEEE 754 standard. The probability is computed by examining how frequently that particular bit flips for every single weight to which stochastic parameter mutation is applied.

In both cases, most of the value bits have a probability greater than 50% of flipping, while the exponent bits (aside from the 3 least significant bits with < 5% probability) and the sign bits do not flip. This is fully anticipated due to the relatively small upper bound on the shift threshold (< 10%), and so we define the HD with respect to the *significand*, which is the portion of the network impacted by the SPM process. As a result, Figure 5(a) shows that the HD increases logarithmically proportional to t_p, and approaches 50% for the significand bits. What stands out in these results, is that in the 32-bit network, even shifting the weights by 1% of their value results in over 40% HD between the resulting networks. This shows that we are able to achieve the desired *variation* between models in the ecosystem without significantly adjusting their weights. This is beneficial because as weights are adjusted by increasingly large amounts (i.e. using a higher t_p), the mutated models are more likely to deviate from their base functionality and thus become unusable.

5.2.2 Ecosystem Performance. With respect to evaluating the relationship between the original model and the ecosystem, we should ensure that the difference in test accuracy of the ecosystem is minimized as much as possible (0, ideally). Hence, we evaluate the *ecosystem performance* on both overall / average accuracy, as well as per-class accuracy. By considering the per-class accuracy, we ensure that the mutation process does not have a harsh negative consequence on a *singular* class significantly over others.



(a) (b) (c)

Fig. 5. (a) Average hamming distance between model weights before and after stochastic parameter mutation is applied in both 32-bit and 16-bit precision. (b-c) Probability of flip for each bit in the weight value in 32-bit and 16-bit precision at different values of t_p. The dotted line marks the final value bit for the corresponding floating point representation (2^{21} for 32-bit and 2^9 for 16-bit). Everything to the right of the dotted line is an exponent or sign bit. The sign bit never flips, and only the 3 least significant exponent bits ever flip.

Figures 6(a-b) show the distribution of the accuracy across each ecosystem. In Figure 6(c) we show a graph of the average accuracy of the ecosystem that was generated for the corresponding value of t_p. In the graph, the cutoff value at 99.0 is the base accuracy, so anything under this line is an undesirable result. With increasing values for t_p there is a gradual loss in accuracy, but it is relatively small. Even for a high value, e.g. $t_p = 10\%$, the loss in accuracy is less than 0.02%. Despite this, we do not want the ecosystem to have a drop in accuracy from the original, so we select $t_p = 0.05$ as the optimum where the level of diversity is maximised, but the accuracy is at or above the baseline. Observing $t_p = 0.05$ as our optimum, we use this as our cutoff value for our experiments we conduct to measure the effect when poisoning the ecosystem in order to minimize computation.

To evaluate the ecosystem performance for larger networks, we applied the same analysis to VGG-16 trained on the CIFAR-10 dataset. Results for the overall accuracy on this network are shown in Figure 6(d-e). The VGG model has a large number of weights and additional layers which impact the SPM process. This limited the range of t_p to about 3%, as SPM at higher percentages resulted in a significant reduction in accuracy. However, we were still able to mutate the weights at up to $t_p = 0.02$ without impacting network performance. We present results for VGG in 32-bit format due to the lack of native support for 16-bit weights in the batch normalization layers. Because of the large number of weights and layers, the errors introduced at each layer by SPM may be compounded to the output and result in a larger error. That being said, the steps taken towards reducing the impact on accuracy – i.e. avoiding bias terms, and defining the cutoff range – help alleviate this to a point where we can still reliably apply SPM to larger networks. Moreover, limiting the range of t_p did not adversely impact the efficacy of the SPM process against Trojan transferability. Interestingly, though, is that the accuracy of the ecosystem could not stabilize towards the baseline for $t_p < 0.01$. This is likely just an anomaly due to the nature of how python/tensorflow generates random numbers that are of that magnitude.

Overall, for both LeNet-5 and VGG-16, diversification through SPM does not compromise the functionality of the networks, and the accuracy is nearly identical, both in terms of individual class accuracy, as well as overall accuracy. However, their structure, in terms of the binary representation of the weight values, is significantly different, as indicated by the HD. Hence, it is possible to

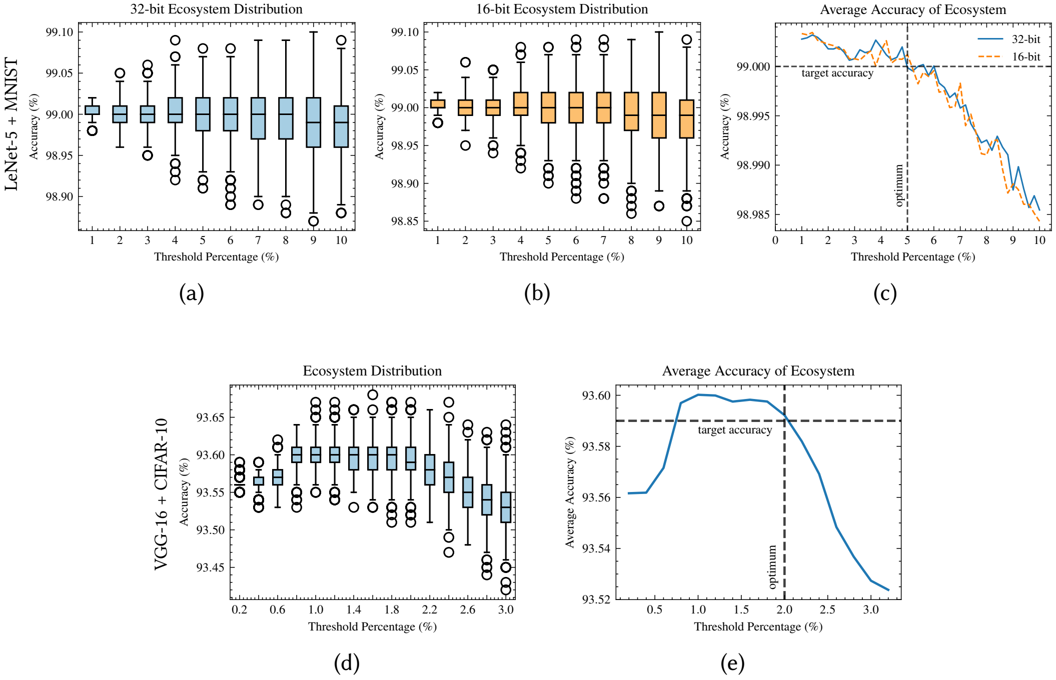


Fig. 6. Average accuracy results for two neural network ecosystems (n = 1000). (Top) LeNet-5 + MNIST: (a) and (b) show the accuracy distribution of the ecosystem for 32-bit and 16-bit networks, and (c) plots the average accuracy for $1.0\% \leq T_p \leq 10.0\%$. (Bottom) VGG + CIFAR-10: (d) shows the accuracy distribution and (e) shows the average accuracy ($0.2\% \leq T_p \leq 3.2\%$).

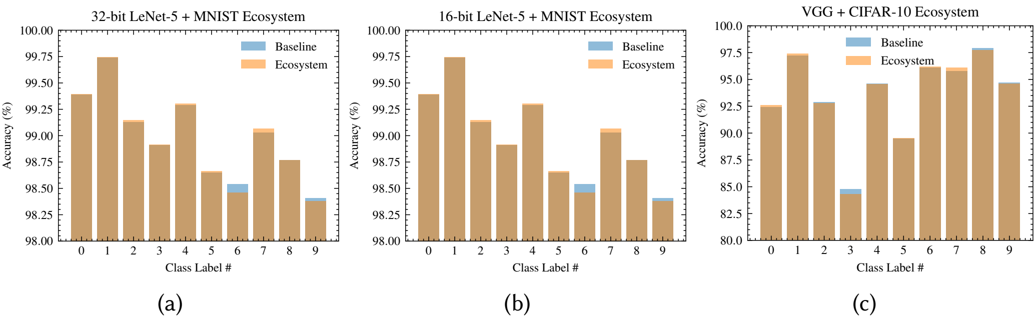


Fig. 7. Average accuracy of each ecosystem broken down into each class. (a-b) Show results for the LeNet-5 + MNIST classifier. (c) Show results for the VGG + CIFAR-10 classifier.

diversify the ecosystem of models without compromising the integrity of the components. Next, we demonstrate the security aspects of our approach.

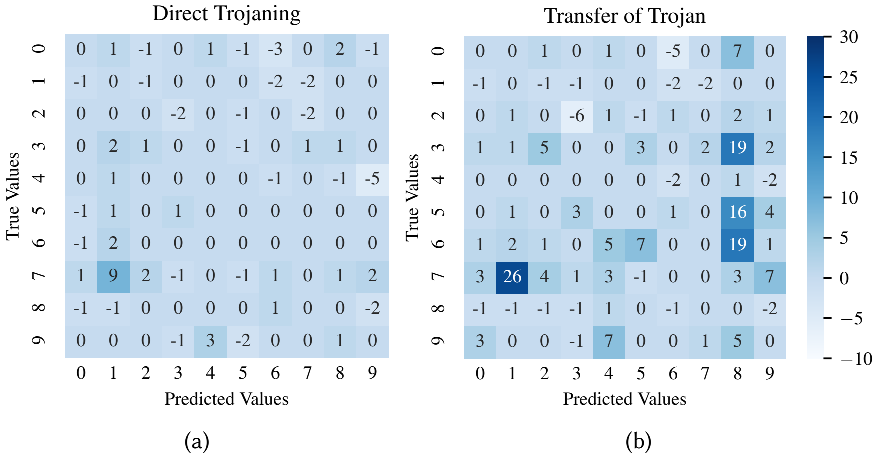


Fig. 8. (a) Confusion matrix showing the difference in classification error of a model before and after the model has been poisoned. (b) Similar confusion matrix showing difference in classification error before and after poisoning, but the poisoning method is done *indirectly* by using an update generated from the model in (a) to another arbitrary model in the ecosystem. For both confusion matrices, positive values indicate *more* errors in those particular classes, while negative values indicate *less* errors.

5.3 Transferability of Neural Trojan

One of the main ideas of this paper is that, given an ecosystem of models that all perform the same function, the success of an attacker's attempts to Trojan a given network in the ecosystem does not guarantee success if they were to attempt to upload the Trojan to another model in the ecosystem. As described in Section 2, a Trojan relies on being as stealthy and undetectable as possible. As such, the given Trojan may only be classified as effective (stealthy) if if it does not affect the accuracy of the model.

In Figure 8, we see two different confusion matrices (a-b) corresponding to two separate models, M_i and M_j, respectively, in the Lenet-5 + MNIST ecosystem. All models in this ecosystem, including M_j, were generated using $t_p = 0.05$. Both confusion matrices show the difference in performance of the model before and after the delta-update procedure. Negative values indicate that the model made less decisions for that respective prediction and true value pairing, while positive values indicated more decisions for that pairing. Essentially, positive values are a deterioration, while negative values are an improvement. In this example, the attacker has inserted a Trojan into M_i which marginally increases the error rate of class 1 to 7. Overall, the accuracy of the model is unchanged; so we can say that this Trojan is successful because it achieves the attacker's goal of targeted misclassification of 1 to 7, while remaining below the detection threshold. The attacker then uploads the Trojan'ed update to another model in the ecosystem, M_j, which results in the confusion matrix in Figure 8(b). As a result of the combined usage of stochastic parameter mutation and delta-update, the stealthiness of the Trojan is lost when attempting to infect another model in the ecosystem.

As mentioned in the experimental setup, we elected to use 4 different configurations for the hyperparameters used when retraining (poisoning) the network. For each configuration, an ecosystem with $n = 1000$ is generated for $t_p = \{0.010, 0.012, 0.014, ...0.050\}$. For each value of t_p, we take the ecosystem and sample 30 random models from it. For each sampled model, we perform the

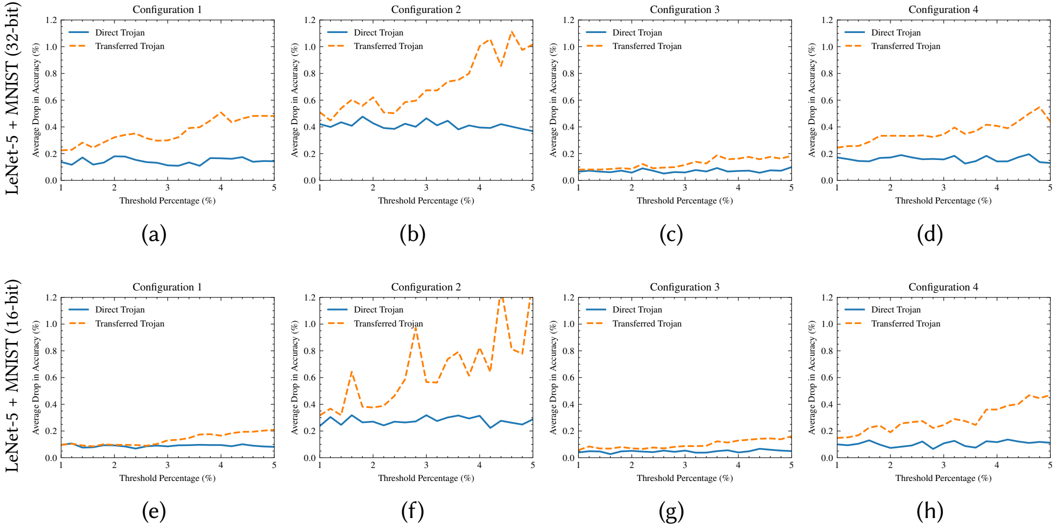


Fig. 9. Graphs showing the average drop in accuracy as a result of mounting the LFA on the LeNet-5 + MNIST classifier, and transferring the attack to other networks in the ecosystem. The top row shows the results for 32-bit networks, and the bottom shows results for 16-bit networks.

poisoning attack, then generate a Trojan'ed update that would be used for that model and apply that update to all of the other models in the ecosystem. The results for those experiments are shown in Figure 10 with (a-d) depicting results for the 32-bit networks and (e-h) depicting results for the 16-bit networks.

The results are quite interesting, and paint a clear picture of the effects as a direct result of the mutation and delta-update procedures. First, we can order the configurations by the amount they change the weights – and thus, accuracy – in ascending order $\{C_3, C_1, C_4, C_2\}$. It is clear that the hyperparameters for configuration 2 ($lr = 5e-3$, batch size = 128, epochs = 50) result in the largest change for the weights, which is expected. This is indicated by the average drop in accuracy of 0.4% for the targeted model. While this configuration may not be ideal for the attacker, it shows what the result would be for the ecosystem in the presence of a poisoning attack that significantly alters the weights. For the LeNet-5 + MNIST ecosystem, among all of the configurations, the drop in accuracy for the ecosystem can be 2-3 times greater than that for the model that was attacked directly.

For the larger VGG-16 model, we performed the same experiments, but only at the optimum threshold for that network ($t_p = 0.02$) in order to save time on computation. The results for these experiments are shown in Figure 10(i-l). Results are comparable to the smaller LeNet-5 + MNIST classifier. For hyperparameter configurations 1-2, we see a significant drop in accuracy as a result of transferring the Trojan, while the same phenomena is less apparent in configurations 3-4. When directly comparing the results between the small and large models, we see that the drops in accuracy are much more significant with the large network. We believe that this is due to the sheer amount of parameters and layers present in the network. Errors introduced by mismatched/malicious updates are compounded through each layer, and ultimately translate to a higher total accuracy drop – worse than a random classifier in the case of configuration 2.

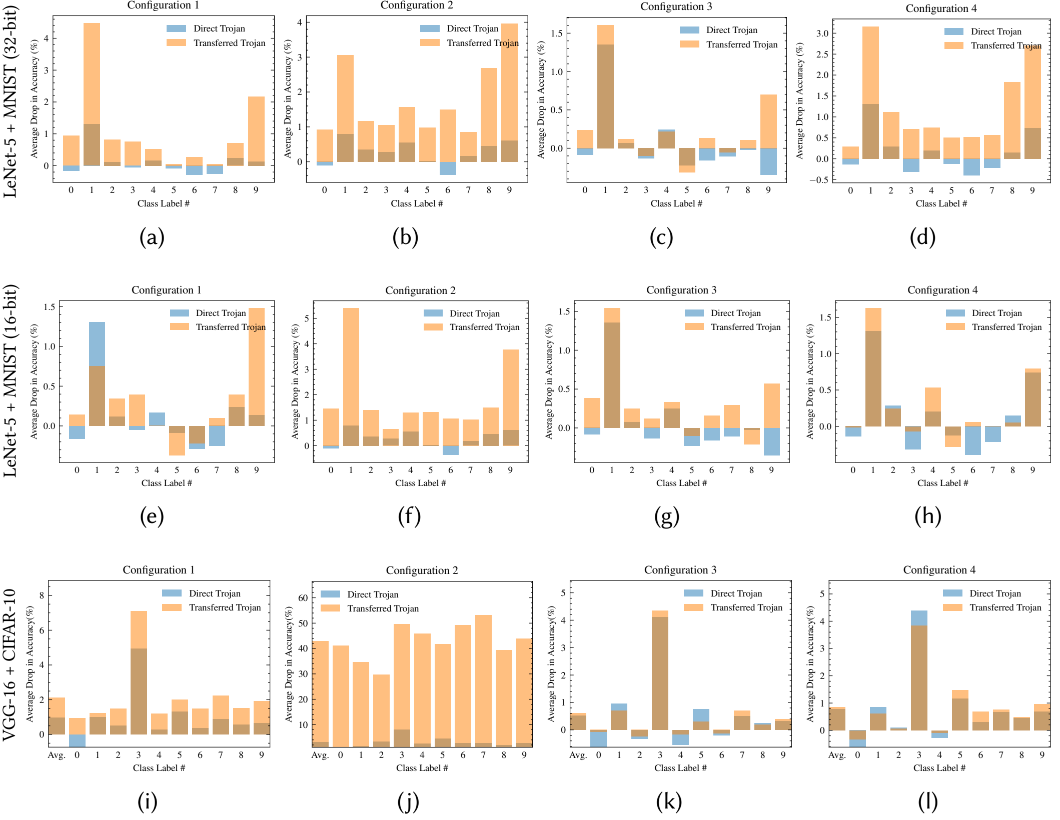


Fig. 10. Graphs showing the average drop in accuracy as a result of mounting the LFA on the given classifier, and transferring the attack to other networks in the ecosystem. The first two rows correspond to the LeNet-5 + MNIST classifier, which uses a threshold of $t_p = 0.05$. The bottom row corresponds to the VGG-16 + CIFAR-10 classifier, which uses a threshold of $t_p = 0.02$.

While the mean absolute change in accuracy between the direct and transferred Trojan does not always appear large, our experiments are conducted across an ecosystem of 1000 models, with 30 sampled individually for direct Trojaning. Thus, each data point on the Transferred Trojan lines in Figure 10 is representative of almost 30,000 data samples. We can prove that the change between the two is statistically significant, and thus would indicate an issue with the update. For instance, let us take the *worst case* for each model (i.e. lowest change between direct/transferred); in all cases, this was configuration 3. For each model, the mean (μ) for effect of direct and transferred Trojans are as follows: LeNet-5 + MNIST (32-bit) $\longrightarrow$ $Direct(\mu = 0.099 \pm 0.130)$ vs. Transferred($\mu = 0.184 \pm 0.137$), LeNet-5 + MNIST (16-bit) $\longrightarrow$ $Direct(\mu = 0.050 \pm 0.038)$ vs. Transferred($\mu = 0.16 \pm 0.080$), and VGG-16 + CIFAR-10 $\longrightarrow$ $Direct(\mu = 0.522 \pm 0.074)$ vs. Transferred($\mu = 0.603 \pm 0.086$). In each of these scenarios, the computed p-value is < 0.0001, which shows that the change is consistently significantly different from the direct Trojan.

6 SECURITY ANALYSIS

In our proposed method, we *do not* claim to thwart an attacker attempting to Trojan a given network; there are a number of other works targeting this in the literature [36, 16, 3]. We instead focus on ensuring that the attacker cannot successfully transfer their Trojan to another device containing the same model, vastly limiting the scale and scope of the attack. Additionally, our approach ensures that, if an attacker is snooping and gains information about a particular update for a network, that this information does not help them attack any other device in the ecosystem. Thus, the remainder of the security analysis of our approach focuses on the attacker's ability to generate an update that can generalize to multiple networks in the ecosystem, while only having access to the model they have mounted their attack on.

Let us assume that the attacker has access to some device that is running a model from the ecosystem, and that they have successfully mounted a Trojan and computed an update file for that model. We will denote the Trojan'ed model as M_t, its update as U_p, and another arbitrary model in the ecosystem as M_i. The difficulty for the attacker arises when they wish to apply U_p to one or more M_i. The exact difficulty is hard to compute, as it involves defining some threshold for the floating point values of the weights and determining some way to adjust the values for Δw in U_p to account for this. However, we can model this difficulty by considering: 1) how many weights are in the network, W, 2) the floating point precision of the weights f_i (specifically the significands), and 3) the accuracy of the model. Using these values, we can formulate the difficulty d as Eqn. 1.

$$d = W * opt(perturb(f_{val})) \tag{1}$$

For every weight w_i in the network, the attacker must compute some value(s) that achieves their goal of activating the respective neuron(s) for the Trojan trigger, while ensuring that the accuracy of the model is not negatively impacted. The complexity of this problem increases relative to the network size and precision of weights used. Further, with a large ecosystem, there is no guarantee that their malicious update will be effective across other devices. The reason that there is one-to-one mapping for an update to its model is because the XOR procedure of the weights can end up shifting the weights by a large, unintended amount. When this occurs, we see the phenomena shown by the confusion matrices in Figure 8 and the graphs in Figure 10, where the impact of the Trojan is much higher than intended by the attacker, and is thus far less stealthy and easier to detect. Empirically, out of 1000 networks, 85%, when updated with U_p, saw at least $2x$ decrease in accuracy over M_t, which would be detected by a BIST consisting of 10,000 test images. Hence, on an individual basis, the victims can identify the issue and take appropriate action, e.g. rolling back to a baseline network configuration. Moreover, by pooling the post-update accuracy loss from the ecosystem as a whole, the overall impact would be obvious, enabling the manufacturer to take ecosystem-wide corrective action as appropriate.

As a practical example of a threat scenario, consider an attacker that aims to circumvent the BIST consisting of N test images. If the attacker Trojans the model, and the accuracy drops by 1%, then we can expect a BIST with $N = 100$ test images to catch the Trojan, as there would be a difference of 1 additional image that was classified incorrectly. Catching this discrepancy is simple if we have the entire test dataset and time to process all the test images. However, depending on the target application, we may not be able to store the entire testing set for quick access in a BIST, and instead have a fraction of the test data on-hand, or only have time to test a fraction of the data, e.g. during a startup process. If we consider a trial with the MNIST classifer where the Trojan degrades the accuracy by 0.15%, then we would expect to catch it if we have a test set of at least $N = 667$ (10,000 / 15) images. For this same Trojan applied to the ecosystem, the accuracy drops by

0.5% – meaning a misclassification would be expected every $N = 200$ (10,000 / 50) images. Because of this, the Trojan-inserted network would fail the BIST for most other devices in the ecosystem.

7 CONCLUSION & FUTURE WORK

In this paper, we presented a novel approach to machine learning security that focuses on protecting an *ecosystem* of variant devices by diversifying configurations which can expose attacks. We presented an approach of *stochastic parameter mutation* which allows IP owners to deploy their model as a diverse *ecosystem* of architecturally different but functionally equivalent variants. This aspect of diversity ensures that, in the event where an attacker is able to successfully mount an attack on one device (model), they will be unable to translate that same attack to another device. The XOR-based update mechanism *delta-update* with the BIST also enables developers to securely update their models in the field.

First, we evaluated our approach using a small, baseline classifier based on LeNet-5, trained on MNIST. Using this toy example, we applied varying levels of SPM and evaluated the ecosystem in terms of performance and resilience – which proved to be very successful. We then scaled our approach up to a much larger model, VGG-16, which was fine-tuned on the CIFAR-10 dataset. Our results for the VGG-16 network showed similar results to that of the baseline model, and, even, that our approach scales *quite well* to larger models. The final error on the output as a result of the malicious update being applied by our system can be *much* larger with larger models. While we achieved excellent results on the VGG-16 network, the threshold for SPM had to be set lower than with the LeNet-5 network, which makes sense. In the future, it would be interesting to investigate different means of implementing SPM on larger networks to allow higher thresholds and thus better security. In general, this paper lays out the intuition of our approach, as well as the foundation for a flexible toolflow to adopt this approach in the field.

REFERENCES

[1] Ateniese, G., Mancini, L. V., Spognardi, A., Villani, A., Vitali, D., and Felici, G. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *Int. J. Secur. Netw. 10*, 3 (Sept. 2015), 137–150.

[2] Bagdasaryan, E., Veit, A., Hua, Y., Estrin, D., and Shmatikov, V. How to backdoor federated learning, 2019.

[3] Baracaldo, N., Chen, B., Ludwig, H., Safavi, A., and Zhang, R. Detecting poisoning attacks on machine learning in iot environments. In *2018 IEEE International Congress on Internet of Things (ICIOT)* (2018), pp. 57–64.

[4] Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanovic, D., and Zovi, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, Association for Computing Machinery, p. 281–289.

[5] Bhagoji, A. N., Chakraborty, S., Mittal, P., and Calo, S. Analyzing federated learning through an adversarial lens, 2018.

[6] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases* (Berlin, Heidelberg, 2013), H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, Eds., Springer Berlin Heidelberg, pp. 387–402.

[7] Biggio, B., Nelson, B., and Laskov, P. Poisoning Attacks against Support Vector Machines. *arXiv* (Jun 2012).

[8] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. End to End Learning for Self-Driving Cars. *arXiv* (Apr 2016).

[9] Chen, X., Liu, C., Li, B., Lu, K., and Song, D. Targeted backdoor attacks on deep learning systems using data poisoning, 2017.

[10] Chollet, F., et al. Keras. https://keras.io, 2015.

[11] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (USA, 2006), USENIX-SS'06, USENIX Association.

[12] Cui, G., Qin, L., Wang, Y., and Zhang, X. An encryption scheme using dna technology. In *2008 3rd International Conference on Bio-Inspired Computing: Theories and Applications* (2008), pp. 37–42.

[13] Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C., and Roli, F. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks, 2018.

[14] DiMattina, C., and Zhang, K. How to modify a neural network gradually without changing its input-output functionality. *Neural computation 22* (10 2009), 1–47.

[15] Fink, G. A., Haack, J. N., McKinnon, A. D., and Fulp, E. W. Defense on the move: Ant-based cyber defense. *IEEE Security Privacy 12*, 2 (2014), 36–43.

[16] Gao, Y., Xu, C., Wang, D., Chen, S., Ranasinghe, D. C., and Nepal, S. Strip: A defence against trojan attacks on deep neural networks, 2020.

[17] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples, 2015.

[18] Graves, A., Mohamed, A., and Hinton, G. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (May 2013), pp. 6645–6649.

[19] Gu, T., Dolan-Gavitt, B., and Garg, S. Badnets: Identifying vulnerabilities in the machine learning model supply chain, 2019.

[20] Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. *CoRR abs/1506.02626* (2015).

[21] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016), pp. 770–778.

[22] Hoque, S., Fairhurst, M., and Howells, G. Evaluating biometric encryption key generation using handwritten signatures. In *2008 Bio-inspired, Learning and Intelligent Systems for Security* (2008), pp. 17–22.

[23] Hu, X., Liang, L., Li, S., Deng, L., Zuo, P., Ji, Y., Xie, X., Ding, Y., Liu, C., Sherwood, T., and et al. Deepsniffer. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar 2020).

[24] Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., and Tygar, J. D. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence* (New York, NY, USA, 2011), AISec '11, Association for Computing Machinery, p. 43–58.

[25] Ji, Y., Zhang, X., and Wang, T. Backdoor attacks against learning systems. In *2017 IEEE Conference on Communications and Network Security (CNS)* (2017), pp. 1–9.

[26] Karam, R., Hoque, T., Ray, S., Tehranipoor, M., and Bhunia, S. Robust bitstream protection in FPGA-based systems through low-overhead obfuscation. *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (Nov 2016), 1–8.

[27] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition* (June 2014), pp. 1725–1732.

[28] Keromytis, A. D. Characterizing software self-healing systems. In *Computer Network Security* (Berlin, Heidelberg, 2007), V. Gorodetsky, I. Kotenko, and V. A. Skormin, Eds., Springer Berlin Heidelberg, pp. 22–33.

[29] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[30] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE 86*, 11 (Nov 1998), 2278–2324.

[31] Li, W., Yu, J., Ning, X., Wang, P., Wei, Q., Wang, Y., and Yang, H. Hu-fu: Hardware and software collaborative attack framework against neural networks. pp. 482–487.

[32] Liao, C., Zhong, H., Squicciarini, A., Zhu, S., and Miller, D. Backdoor embedding in convolutional neural network models via invisible perturbation, 2018.

[33] Liu, T., Wen, W., and Jin, Y. Sin2: Stealth infection on neural network — a low-cost agile neural trojan attack methodology. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2018), pp. 227–230.

[34] Liu, Y., Chen, X., Liu, C., and Song, D. Delving into transferable adversarial examples and black-box attacks, 2016.

[35] Liu, Y., Ma, S., Aafer, Y., Lee, W.-C., Zhai, J., Wang, W., and Zhang, X. Trojaning attack on neural networks. In *25nd Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-221, 2018* (2018), The Internet Society.

[36] Liu, Y., Xie, Y., and Srivastava, A. Neural Trojans. *arXiv* (Oct 2017).

[37] Mahmud, S., Olney, B., and Karam, R. Architectural diversity: Bio-inspired hardware security for fpgas. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)* (2018), pp. 48–51.

[38] Moosavi-Dezfooli, S., Fawzi, A., Fawzi, O., and Frossard, P. Universal adversarial perturbations. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 86–94.

[39] Muñoz-González, L., Biggio, B., Demontis, A., Paudice, A., Wongrassamee, V., Lupu, E. C., and Roli, F. Towards

poisoning of deep learning algorithms with back-gradient optimization, 2017.
[40] Olney, B., and Karam, R. Tunable FPGA Bitstream Obfuscation with Boolean Satisfiability Attack Countermeasure. *ACM Trans. Des. Autom. Electron. Syst. 25*, 2 (Feb 2020), 1–22.
[41] Opitz, D. W., and Shavlik, J. W. Generating accurate and diverse members of a neural-network ensemble. In *Proceedings of the 8th International Conference on Neural Information Processing Systems* (Cambridge, MA, USA, 1995), NIPS'95, MIT Press, p. 535–541.
[42] Papernot, N., McDaniel, P., and Goodfellow, I. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples, 2016.
[43] Rakin, A. S., He, Z., and Fan, D. Tbt: Targeted neural network attack with bit trojan. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 13195–13204.
[44] Rathore, H., and Jha, S. Bio-inspired machine learning based wireless sensor network security. In *2013 World Congress on Nature and Biologically Inspired Computing* (2013), pp. 140–146.
[45] Reith, R. N., Schneider, T., and Tkachenko, O. Efficiently stealing your machine learning models. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2019), WPES'19, Association for Computing Machinery, p. 198–210.
[46] Rolnick, D., Veit, A., Belongie, S., and Shavit, N. Deep Learning is Robust to Massive Label Noise. *arXiv* (May 2017).
[47] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2015.
[48] Song, C., Ristenpart, T., and Shmatikov, V. Machine learning models that remember too much, 2017.
[49] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. In *International Conference on Learning Representations* (2014).
[50] Tehranipoor, M., and Koushanfar, F. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Des. Test Comput. 27*, 1 (Feb 2010), 10–25.
[51] Tramèr, F., Papernot, N., Goodfellow, I., Boneh, D., and McDaniel, P. The space of transferable adversarial examples, 2017.
[52] Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. Stealing machine learning models via prediction apis, 2016.
[53] Wang, B., Yao, Y., Shan, S., Li, H., Viswanath, B., Zheng, H., and Zhao, B. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. pp. 707–723.
[54] Wang, R., Zhang, G., Liu, S., Chen, P.-Y., Xiong, J., and Wang, M. Practical detection of trojan neural networks: Data-limited and data-free cases, 2020.
[55] Xiao, H., and Eckert, C. Adversarial label flips attack on support vector machines. 870–875.
[56] Yi Shi, Sagduyu, Y., and Grushin, A. How to steal a machine learning classifier with deep learning. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)* (2017), pp. 1–5.
[57] Yu, H., Ma, H., Yang, K., Zhao, Y., and Jin, Y. Deepem: Deep neural networks model recovery through em side-channel information leakage.
[58] Yu, H., Yang, K., Zhang, T., Tsai, Y.-Y., Ho, T.-Y., and Jin, Y. Cloudleak: Large-scale deep learning models stealing through adversarial examples.
[59] Zareen, F., and Karam, R. Detecting RTL Trojans using Artificial Immune Systems and High Level Behavior Classification. *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)* (Dec 2018), 68–73.
[60] Zeiler, M. D., and Fergus, R. Visualizing and understanding convolutional networks. *CoRR abs/1311.2901* (2013).
[61] Zhao, P., Chen, P.-Y., Das, P., Ramamurthy, K. N., and Lin, X. Bridging mode connectivity in loss landscapes and adversarial robustness, 2020.