Clemson University

TigerPrints

All Dissertations | Dissertations

5-2023

Adversarial Deep Learning and Security with a Hardware Perspective

Joseph Clements
jfcleme@g.clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

 Part of the Other Computer Engineering Commons

Recommended Citation

Clements, Joseph, "Adversarial Deep Learning and Security with a Hardware Perspective" (2023). *All Dissertations*. 3352.
https://tigerprints.clemson.edu/all_dissertations/3352

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

Adversarial Deep Learning and Security with a Hardware Perspective

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Joseph Clements
May 2023

Accepted by:
Dr. Yingjie Lao, Committee Chair
Dr. Long Cheng
Dr. Richard Groff
Dr. Adam Hoover
Dr. Apoorva Kapadia

Abstract

Adversarial deep learning is the field of study which analyzes deep learning in the presence of adversarial entities. This entails understanding the capabilities, objectives, and attack scenarios available to the adversary to develop defensive mechanisms and avenues of robustness available to the benign parties. Understanding this facet of deep learning helps us improve the safety of the deep learning systems against external threats from adversaries. However, of equal importance, this perspective also helps the industry understand and respond to critical failures in the technology. The expectation of future success has driven significant interest in developing this technology broadly. Adversarial deep learning stands as a balancing force to ensure these developments remain grounded in the real-world and proceed along a responsible trajectory. Recently, the growth of deep learning has begun intersecting with the computer hardware domain to improve performance and efficiency for resource constrained application domains. The works investigated in this dissertation constitute our pioneering efforts in migrating adversarial deep learning into the hardware domain alongside its parent field of research.

This work covers two novel perspective: hardware Trojans and hardware watermarks for deep learning hardware accelerators. A foundational work for both of these perspectives is the operational backdoor. Backdoor injection in deep learning compromises an intelligent systems by modifying the system during development to introduce abnormal behaviors which can be activated by the adversary upon deployment. The operational backdoor uses a fundamentally different vulnerability than previous methodologies. Conventional backdoors in deep learning exploits the ability of the adversary to inject changes in model parameters, architectures, or training procedures. Our method exploits the vulnerability of deep learning to modifications to its fundamental computational operations, such as the non-linear activation functions. This unique perspective on injecting backdoors enables an adversary to compromise a model through deep learning programming frameworks,

firmware updates, or faults in the computational circuits.

Then, we extend the potential of hardware Trojan attacks in the domain of adversarial deep learning. Hardware Trojans describe modifications to hardware circuitry which inject some malicious functionality into a hardware design. Modern trends in the production and distribution of hardware components along a globalized supply chain are highly beneficial to the industry and largely enable the widespread availability of computing devices. However, the ease of access to a hardware design and divergent incentives make Hardware Trojans a genuine possibility for those same computing platforms. Further, the immutability of hardware and the difficulty of identifying Trojan functionality makes mitigation of such attacks an unsolved problem in the industry. Our research finds that hardware Trojans can be injected into a hardware platform that is able to compromise a deep learning model executed on the platform.

Recently, the ease with which an adversary is able to pirate high-value deep learning models has become a serious concern. Multiple avenues to defend against this vulnerability have been extended to deep learning from other domains. Watermarks are one such technique that embeds a signature into the deep learning model such that fraudulent usage of the model can be detected and remedied. This defense is typically the last line of defense against piracy and so is a critical tool for deep learning developers. As we incorporate the hardware perspective into deep learning, protecting high-value deep learning hardware from piracy secures the profit incentives and creative endeavors that motivate future developments. We tackle this problem in our work by developing a sophisticated watermarking framework that utilizes a hardware-algorithmic cooperative algorithm that enables a deep learning hardware developer to inject watermarks into their designs, identify piracy, and support claims of fraudulent usage.

Dedication

To my wonderful and loving wife, Rose. Thank you for all your support!

Acknowledgments

I would like to express my gratitude and appreciation to my adviser Dr. Yingjie Lao for his guidance and encouragement throughout this work.

I further extend my gratitude to each of my committee members: Dr. Adam Hoover, Dr. Richard Groff, Dr. Apoorva Kapadia, and Dr. Long Cheng, for their time and interest in this work. I want to acknowledge Dr. Adam Hoover's gracious support in helping me develop my professional writing and presentation skills.

I would finally like to acknowledge the faith and support of my family and the members of my research group.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work explores the field of adversarial deep learning from the hardware perspective by evaluating the effect of manipulations in the hardware domain on deep learning systems. Conventional works in adversarial deep learning have predominantly focused on manipulations to the deep learning system directly in the software. We introduce the hardware perspective into adversarial deep learning with pioneering works exploring the effectiveness of hardware Trojans against deep learning systems and the efficacy of embedding watermarks into deep learning hardware accelerators. Through this effort, we find that a hardware-aware perspective of the study of adversarial deep learning presents novel vulnerabilities and defensive capabilities that are not present from the software perspective alone.

1.1 The Impact of Deep Learning

Conventional computing practices have long relied on humans with intimate knowledge of a target application to produce programs to accomplish a given task. We can understand the task these programs perform as a transformation $F : \mathcal{X} \rightarrow \mathcal{Y}$ from an input domain, $\mathcal{X}$, to an output domain, $\mathcal{Y}$. Machine learning challenges this paradigm by creating algorithms that allow mathematical models, which can be denoted as $F(\phi, \cdot)$, with trainable parameters, ϕ, to adapt to observed data. ϕ is trained by applying a training dataset of N_t data-points, $\mathcal{D}_T = \{\mathbf{x}_t, \mathbf{y}_t\}^{N_t} \in \{\mathcal{X}, \mathcal{Y}\}$. A loss function, $\mathcal{L}$, which quantifies the error, or difference, between two elements of $\mathcal{Y}$, can be used to define an optimization problem. The simplest optimization problems seek to minimize the loss over

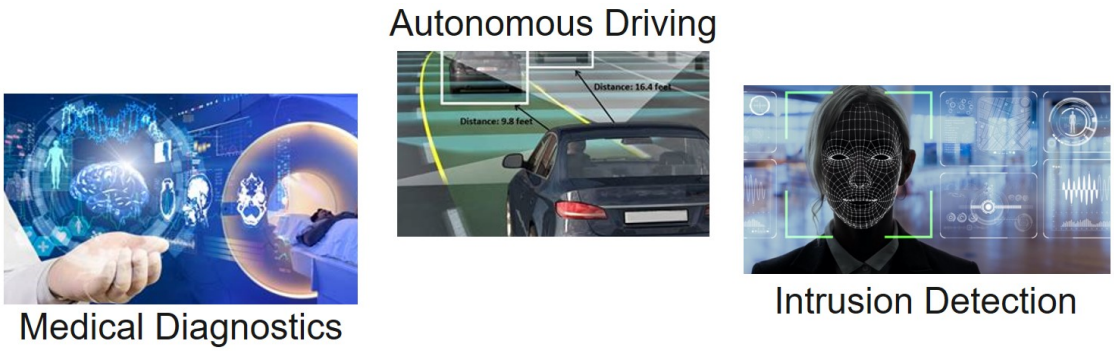

Figure 1.1: Deep learning is enabling state-of-the-art advancements in many high-profile domains, but adversarial manipulation could result in catastrophic failure, especially in security critical settings.

the training dataset.

$$\min_{\phi} \sum_{\mathbf{x}_t, \mathbf{y}_t \in \mathcal{D}_T} \mathcal{L}(F(\phi, \mathbf{x}_t), \mathbf{y}_t) \tag{1.1}$$

Such problems are solved using gradient descent-based algorithms. This foundational understanding can be extended to facilitate constraints and secondary objectives for specific applications.

Deep neural networks are a potent family of mathematical models used in machine learning. Deep neural networks are composed of several layers of parallel computational units called neurons. Each neuron performs the operation: $f(\sum_{j=1}^{N_l} h_j \times w_j + b)$ where f is a non-linear activation function. h_j is the j-th element of the previous layer's outputs with corresponding weight w_j, and b is a trainable bias. Each layer is understood as performing a linear transformation of the layer's inputs and scaling them through the non-linear activation function. Individual layers are then arranged into a graph structure where the previous layer's outputs are fed as input to subsequent layers. These models are called "deep" because of the multi-layered structure that incorporates multiple layers of abstraction into the model. Machine learning systems built on a deep model are distinguished as deep learning [58].

Deep learning has become an incredibly potent tool for powering modern technologies. Though still in its infancy, deep learning has seen remarkable success in computer vision [118], natural language processing [162], bio-informatics [163], and cybersecurity [40]. This technology has driven critical breakthroughs in historically challenging fields like autonomous driving [33], manufacturing [84], nuclear power [126], and medical imaging [13]. However, many of the prominent

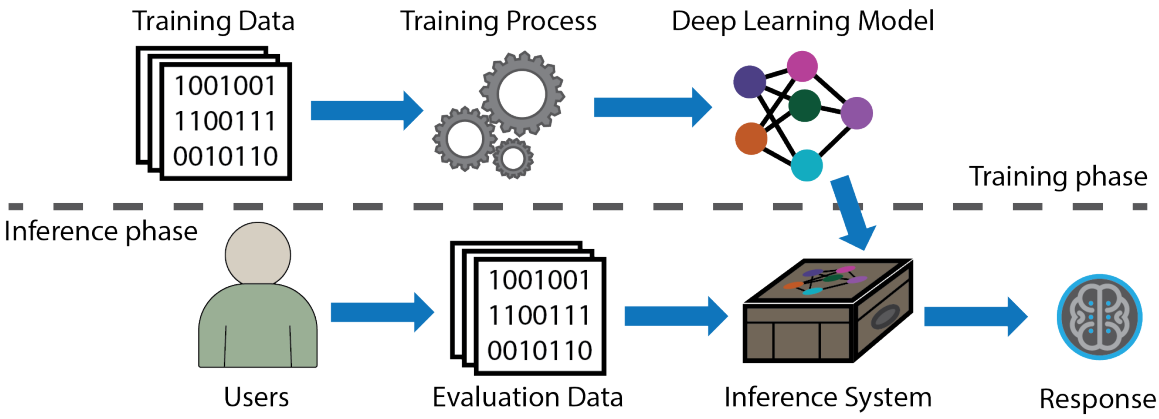

Figure 1.2: The development of deep learning systems is typically described as being composed of *training* and *inference* phases. This perspective reflects the trends observed in modern cloud-based or deep learning as a service paradigms.

application domains are safety-critical, and faults in the deep learning systems could result in significant harm. As such, the robustness of deep learning systems in such settings is of critical importance.

The conventional perspective of deep learning considers two phases in the development of deep learning systems, a training phase and an inference phase. In the training phase, a training dataset is used as input to a deep learning algorithm in order to produce the deep learning model. During the inference phase, the model is deployed to a system that accepts and processes user input. This reflects the common trend for developers to utilize large high-performance computing clusters to train a deep learning model before deploying it in deep learning as a service setting [125]. A user must then access the deep learning model through remote API calls, transmitting data to the cloud. On the cloud, the data is processed, and the model's response is transmitted back to the user. This perspective has largely directed the understanding of deep learning systems and their adversarial implications. However, this is not optimal for many applications. Autonomous driving, as an example, can be critically impacted by a few seconds of interrupted communications resulting in both the destruction of property and the potential loss of life [98]. Further, security or privacy concerns are inherent in some application domains like healthcare, which has a limited ability to transmit data to third-party entities [142]. These issues and more make migration to on-device computing in deep learning a highly desirable shift in many applications [39].

1.2 Dangers of Adversaries in Deep Learning

Adversarial deep learning is the field of study that explores the security implications by evaluating the capabilities of an adversary with access to a deep learning system [122]. This perspective has led to the discovery of critical vulnerabilities to deep learning systems such as adversarial examples [141] and data poisoning attacks [3]. Some of these vulnerabilities have been met with defensive techniques that mitigate their adversarial usage [155]. However, others continue to be an active area of discussion [68]. In addition to these security implications, it is also understood that understanding the failure modes of deep learning systems is critical to developing systems robust to natural phenomena as well [18].

The current conceptualization that deep learning is primarily composed of training and testing phases has also largely shaped the adversarial perspective. Adversarial deep learning also be classified into two families of attacks: causative and exploratory attacks [147]. A causative attack largely considers an adversary with some control in the training phase of a deep learning system, where the adversary is able to manipulate the generation of a model introducing faults that can be exploited at inference time. While exploratory attacks often visualize an adversary in the inference phase, with little access to the generation of the model. Such an adversary seeks to find faults naturally present in the deep learning system. These two perspectives have led to a large body of interesting research and brought awareness to practical weaknesses in deep learning systems that developers should account for.

1.2.1 Attacks on the Training Phase

1.2.1.1 Data Poisoning Attacks

A typical example of an attack on a deep learning model's training phase is the data poisoning attack. This family of attacks considers a scenario in which the adversary has some control over the training dataset. With such access, he may attempt to alter the data to inject some malicious functionality into a model [101, 25]. We can describe this process mathematically by defining a poisoned dataset, $\mathcal{D}'_T = \{(\mathbf{x}_t, \hat{\mathbf{y}}_t)_{t=1}^{N_{\hat{T}}}\}$ for $\hat{\mathbf{y}}_t$, altered labels. This poisoned dataset can then be injected into the training dataset without the awareness of the developer. Unbeknownst to the developer,

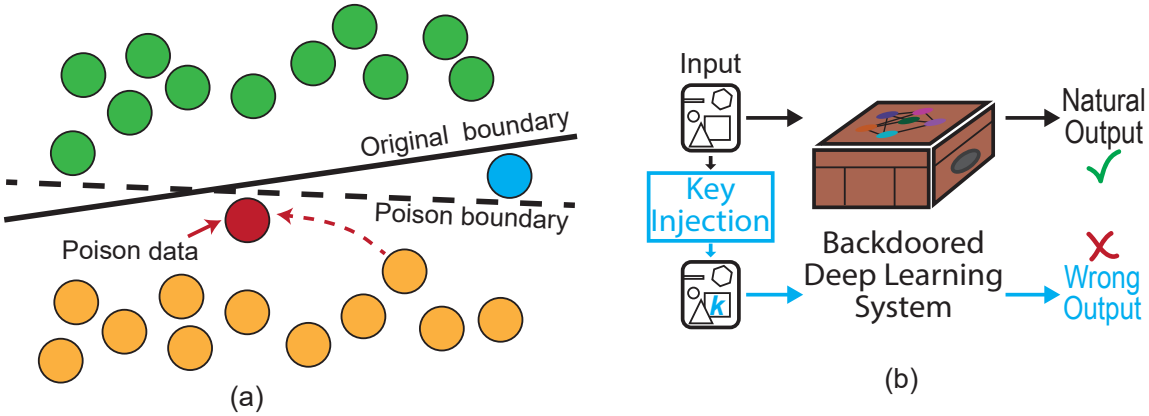

Figure 1.3: Data poisoning (a) alters or injects examples in a training dataset to produce adverse effects on a deep learning system. This can be used to inject backdoors (b), which introduces new functionality to a system that can give an adversary control during inference.

this alters the optimization problem being solved:

$$\min_{\phi} \sum_{\mathbf{x}_t, \mathbf{y}_t \in \mathcal{D}_T} \mathcal{L}(F(\phi, \mathbf{x}_t), \mathbf{y}_t) + C \sum_{\mathbf{x}_t, \hat{\mathbf{y}}_t \in \mathcal{D}'_T} \mathcal{L}(F(\phi, \mathbf{x}_t), \hat{\mathbf{y}}_t) \tag{1.2}$$

where C is a hyper-parameter that describes how the malicious and benign functionalities are balanced in the training process.

Various algorithms for data poisoning attacks have been developed for various attacker constraints and capabilities. In most scenarios, there are basic assumptions that the attack must be stealthy in order to bypass detection. Data/Label manipulation attacks, for example, attempt to alter the inputs/labels of existing training data to poison the dataset rather than introducing novel training examples. Such attacks do not alter the number of training examples and do not trigger trivial checks which track the size of the training dataset [14, 114]. Further, many attacks attempt to minimize the number of alterations to the training dataset. This increases the likelihood that a random subset of the training dataset can be analyzed to reveal the poisoned data [25]. Recently, clean-label attacks have arisen, which have proven to be much stealthier than traditional attacks. Clean label attacks leave the labels accurate to the true label. Such manipulations are difficult to deal with as even careful human inspections do not reveal the manipulated data [74].

1.2.1.2 Deep Learning Backdoors

Backdoors have also been shown to be quite effective in compromising deep neural networks. In the deep learning literature, a backdoor refers to an abnormal functionality in a model which allows a user to specify the output of a deep learning model. During inference, the adversary can apply an input key to activate the backdoor [25, 101, 61]. Adversaries with access to a model's training phase can embed backdoors into the model through various techniques, including data poisoning [25]. Other methodologies have reverse-engineered the training dataset and retrained the model to embed the backdoor while preserving the original functionality [101]. It is even possible for adversaries to inject backdoor functionality through a compromised deep learning framework [8]. While many defense mechanisms have been proposed to confront backdoor attacks [96], novel backdoor attacks continue to find vulnerabilities in machine learning security [93].

We understand the goal of the designer is to train a model, $F(\cdot)$, by minimizing Equation 1.1. Embedding a backdoor can similarly be understood as trying to embed a relation such that

$$\min_{\phi} \sum_{\mathbf{x}_t, \mathbf{y}_t \in \mathcal{D}_T} \mathcal{L}(F(\phi, \mathcal{P}(\mathbf{x}_t, \mathbf{k})), \hat{\mathbf{y}}) \tag{1.3}$$

is also minimized. Here, $\mathcal{P}(\cdot, \cdot)$ is some key embedding process which embeds a key, $\mathbf{k}$ in $\mathbf{x}_t$. A common example uses a pattern/mask pair as the key, mathematically denoted as $\mathbf{k} = (\mathbf{p}, \mathbf{m})$. $\mathbf{p}$ defines a pixel value pattern that is injected into an image by the trigger, while $\mathbf{m}$ defines a region of the input to be overwritten by $\mathbf{p}$.

$$\mathbf{x}'_t = \mathcal{P}(\mathbf{x}_t, (\mathbf{p}, \mathbf{m})) = \mathbf{x}_t \odot (1 - \mathbf{m}) + \mathbf{p} \odot \mathbf{m} \tag{1.4}$$

where $\odot$ represents the element-wise product.

With this trigger, the adversary can define a poisoned dataset $\mathcal{D}'_T = \{\mathbf{x}'_t, \hat{\mathbf{y}}\}_1^{N_T}$. Then, using a data poisoning attack can force the model to minimize the optimization problem:

$$\min_{\phi} \sum_{\mathbf{x}_t, \mathbf{y}_t \in \mathcal{D}_T} \mathcal{L}(F(\phi, \mathbf{x}_t), \mathbf{y}_t) + C \sum_{\mathbf{x}'_t, \hat{\mathbf{y}} \in \mathcal{D}'_T} \mathcal{L}(F(\phi, \mathbf{x}'_t), \hat{\mathbf{y}}). \tag{1.5}$$

Minimizing these problems introduces an association between ($\mathbf{p}$,$\mathbf{m}$) and $\hat{\mathbf{y}}$ to the model while preserving its initial functionality. As such, the model tends to correctly produce the response:

$F(\phi, \mathbf{x}_t) = \mathbf{y}_t$, for natural inputs. However, for the inputs injected with $\mathbf{k}$, the model is likely to predict them as $F(\phi, \mathcal{P}(\mathbf{x}_j, (\mathbf{p}, \mathbf{m}))) = \mathbf{y}_t$.

This concept has been further extended with more developed methods for generating the key-injected inputs. Of particular note, this has resulted in physical world attacks, which allow adversaries to introduce physical objects into images to fool visual detectors [25]. Others have embedded key information in the input wavelet transforms to fool the model [44]. In some cases, even natural images can embed a backdoor in a model. So backdoors can be injected simply through intentional data collection [158].

1.2.2 Attacks on the Inference Phase

1.2.2.1 Adversarial Examples

Adversarial examples have emerged as a research topic of great interest in recent years. These attacks attempt to generate an input that results in adverse classification predictions in a machine learning model while being indistinct from a correctly processed input [146]. Various techniques of adversarial examples have been developed in the literature [4, 149, 4, 112, 164, 18].

The broad family of adversarial example generation algorithms can be described using the unified optimization problem articulated in [18] as:

$$\begin{aligned} \underset{\mathbf{x}'_t}{min} \quad & \mathcal{L}(F(\phi, \mathbf{x}'_t), \hat{\mathbf{y}}) \\ s.t. \quad & S(\mathbf{x}'_t, \mathbf{x}_t) < \epsilon. \end{aligned} \tag{1.6}$$

Where $\mathcal{L}(\cdot, \cdot)$ is a loss function connecting the model's behavior given a modified input, $\mathbf{x}'_t$, and a desired target output, $\hat{\mathbf{y}}$. $S(\cdot, \cdot)$ is a metric that measures $\mathbf{x}'_t$ the similarity of $\mathbf{x}'_t$ to the original input, $\mathbf{x}_t$. If it is possible to optimize $\mathbf{x}'_t$ such that ϵ remains small, an adversarial example is produced. For instance, early attacks in the field, such as the fast gradient sign method (FGSM) [146], and the Jacobian-based saliency maps attack (JSMA) algorithm [122] performed a single iteration of gradient descent to quickly generate adversarial inputs. Following these works, several advanced and iterative methods have been proposed to generalize the attacks or generate stronger adversarial inputs that could mitigate the state-of-the-art defense techniques [124, 7].

A widely used method to solve this problem is through some variations on the projected gradient descent (PGD) algorithm, which is currently considered among the strongest first-order

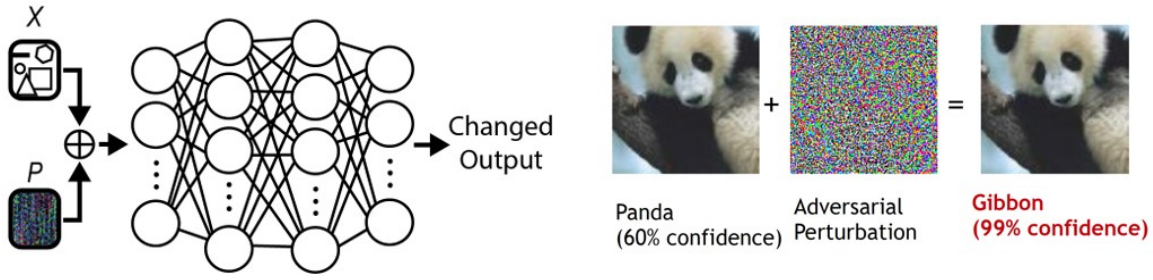

Figure 1.4: Adversarial examples find a perturbation on an input that controls the behavior of deep learning models, despite appearing to be effectively the same to a human observer [59].

attacks. The method is conducted by choosing a step size, α, and iteratively perturbing the input using the perturbation equation:

$$\mathbf{z}_t^{n+1} = \mathbf{x}_t^n + \alpha \bar{\mathbf{s}}_n, \tag{1.7}$$

where $\bar{\mathbf{s}}_n$ is a unit vector in the direction of perturbation at iteration n, i.e., the unit step, as expressed in Equation 1.8. This vector is generated with the gradient of the loss function, $\mathcal{L}$, and a measure of distance, $\mathcal{D}$, often the euclidean distance.

$$\bar{\mathbf{s}}_n = \underset{\mathcal{D}(\mathbf{s})=1}{-argmin} \|\nabla_{\mathbf{x}} \mathcal{L}(F(\phi, \mathbf{x}_t^n), \hat{\mathbf{y}}) - \mathbf{s}\|. \tag{1.8}$$

However, there is no guarantee that the intermediate step, $\mathbf{z}_t^{n+1}$, fulfills the similarity constraint. So *PGD* projects this input back into the set of feasible inputs by solving Equation 1.9.

$$\mathbf{x}_t^{n+1} = \underset{\mathbf{x} \in \mathcal{B}_p(\mathbf{x}_t, \epsilon)}{argmin} \|\mathbf{z}_t^{n+1} - \mathbf{x}\|, \tag{1.9}$$

where $\mathcal{B}(\mathbf{x}_t, \epsilon)$ is a constraint ball of radius with ϵ to bound the perturbation. Most works have used ℓ_p-norms for similarity, thus projecting adversarial steps onto $\mathcal{B}_p(\mathbf{x}_t, \epsilon) = \{\mathbf{x}_t^n \mid \|\mathbf{x}_t^n - \mathbf{x}_t\|_p < \epsilon\}$, an ℓ_p-ball. For this task, the ℓ_∞ [59, 107], ℓ_2 [24], and so-called "ℓ_0" [122] balls appear most frequently. But, such methods frequently produce inputs that differ greatly from the natural inputs, and thus, recent works have begun expanding these concepts to alternate metrics of distance [106, 160].

One of the more interesting aspects of the adversarial example problem is that it has been repeatedly shown that all deep learning systems and inputs are susceptible to adversarial examples to some degree [123]. This leads many researchers to believe adversarial examples indicate a funda-

mental fault in modern deep learning systems [75]. In addition, the adversarial examples generated for one model tends to transfer to similar models enabling adversarial example attacks against deep learning models even in settings where the adversary cannot access the model gradients [100].

1.2.2.2 Model Stealing

The high value of deep learning technologies makes them high-value intellectual properties (IPs), which adversaries have a significant incentive to pirate. As such, many works have explored the possibility of adversaries pirating or stealing deep learning models [150, 151, 41, 136]. In addition to directly stealing the model, the datasets and algorithms used to generate deep learning models can be considered of equivalent value, and so various works have also explored an adversaries capabilities in stealing such IPs [69]. This concern simultaneously raises privacy concerns for deep learning datasets [64].

A common method adversaries can use to pirate deep learning models is through model extraction attacks [150]. In such attacks, the objective of the adversary is to produce a substitute model, $F'(\theta, \cdot)$, with parameters θ which approximates the functionality of the target model $F(\phi, \cdot)$. Trivially, this process can be done with access the predictions of $F(\phi, \cdot)$ and a collection of example inputs, $\mathcal{X}_T = \{\mathbf{x}_t\}_1^{N_t}$. Then, the model parameters, θ, can be found with the optimization problem:

$$\min_{\theta} \sum_{\mathbf{x}_t \in \mathcal{X}_T} \mathcal{L}(F'(\theta, \mathbf{x}_t), F(\phi, \mathbf{x}_t)) \tag{1.10}$$

With unlimited inference access to the target model, $F(\phi, \cdot)$, training the parameters, θ, becomes trivial, and so various methods for limiting an adversaries access to the model's inference results have arisen to obscure deep learning systems from model extraction attacks.

In response to such defenses, algorithms that can extract deep learning models with a minimal number of inference calls have been developed. Such methods have also been applied in adversarial example pipelines to generate substitute models in black box settings [121].

1.2.3 Defending Deep Learning Systems

While a broad range of attacks has been developed against deep learning systems, various defenses for these systems have also been proposed in the deep learning literature. Unfortunately, many of these vulnerabilities are still open problems in modern deep learning.

1.2.3.1 Deep Learning Watermarks

Piracy is a serious concern for deep learning models as these systems are highly-valuable intellectual properties (IPs) but relatively easy to copy. While the model extraction attacks discussed in Section 1.2.2.2 pose a significant threat of piracy to deep learning models, additional vulnerabilities may allow adversaries to directly copy model parameters or reverse engineer datasets making piracy a difficult problem to stop [156]. Some techniques have arisen to proactively defend these systems from such attacks; however, these solutions are not infallible, and it is possible that piracy still occurs despite such defenses [57]. One popular method to defend deep learning systems is to obfuscate the model upon theft [21]. Deep learning watermarks have arisen as a solution that attempts to mitigate the problem of piracy by giving developers an avenue for identifying piracy after it occurs [156].

Of critical importance in watermarking is a verification method, or the process by which the owner can demonstrate the presence of their signature in the mode. DeepSigns, for example, embeds multiple signatures in each layer of a model [34]. The signature in the final layer can be activated in a black box setting to identify the model. Once identified, the model can be accessed in a white box setting, thoroughly proving ownership over the system. While various methods for watermarking deep learning systems have arisen, one of the more common approaches is for deep learning developers to embed intentional backdoors in their IPs which can be used to identify the model [2]. In general, this process can be seen as solving the problem:

$$\min_{\phi} \sum_{\mathbf{x}_t, \mathbf{y}_t \in \mathcal{D}_T} \mathcal{L}(F(\phi, \mathbf{x}_t), \mathbf{y}_t) + C \sum_{\mathbf{x}_k, \hat{\mathbf{y}}_k \in \mathcal{D}_S} \mathcal{L}(F(\phi, \mathbf{x}_k), \hat{\mathbf{y}}_k). \tag{1.11}$$

where $\mathcal{D}_S = \{\mathbf{x}_k, \hat{\mathbf{y}}_k\}_i^{N_s}$ is dataset encoding the owner's signature. In this scenario, $\hat{\mathbf{y}}_k$ should be chosen to be distinct from the natural output, $\mathbf{y}_k$, but should be chosen to be non-adversarial in nature.

1.3 Hardware Perspective of Deep Learning

The current state of deep learning hardware acceleration is to utilize general-purpose processors like graphic processing units (GPUs) [88]. While these processors are significantly more potent than CPUs and enable a significant degree of parallelism, which greatly benefits deep learning models, FPGA [134] and ASIC [108] solutions can boost performance by an order of magnitude

or more. This reality has driven key players in deep learning to develop proprietary dedicated hardware platforms for deep learning applications [79, 35]. These solutions are still designed around general-purpose applications, and only recently have we seen an interest in designs better targeted at specific applications/software-hardware collaborative designs [63]. As the field advances and more players enter the space, we will continue to see a growing interest in novel hardware designs for the field.

While there is a vast number of distinct architectures in deep learning, for the most part, deep learning models all use similar computations: multiplications, additions, max pooling, etc. In fact, most of these operations can be represented as matrix and vector operations making them perfect candidates for specialized hardware. Many key players in the technology industry have already developed notable specialized hardware platforms, including DianNao [28], TrueNorth [36], Eyeriss [27], and Tianjic [38]. Such platforms are still relatively general-purpose and designed to be integrated with broad families of neural network architectures. However, software-hardware collaborative efforts have also been seen, which attempt to specialize hardware alongside a specific target architecture to improve performance for specific applications [113].

A multitude of optimizations can be made during this design flow to improve the trade-offs of a design for a specific purpose. Quantization reduces the bit-width of data in the deep learning model to reduce the time, area, and power required to perform computation [152]. Dataflow schemes can be used to reduce the movement of data in the reducing data transfer requirements [27]. Sparse computations can reduce the overall number of operations hardware computes by dropping unnecessary operations, such as multiplications by 0 [66]. Some of these optimizations, such as quantization, utilize to approximate computations which in some cases can result in different predictions when computed on the hardware over the theoretical model. It is important for developers to be aware of the effect of such hardware optimizations on the system, as often, models can be trained with an awareness of these optimizations to minimize their effects.

While deep learning theory largely considers the training and inference phases to be the major contributors to the development of a deep learning system, the possibility of the hardware impacting the final system introduces new avenues for the security of such systems. This interaction between deep learning and the hardware platforms from the perspective of the system's security has been largely unexplored, and many potential research opportunities still remain.

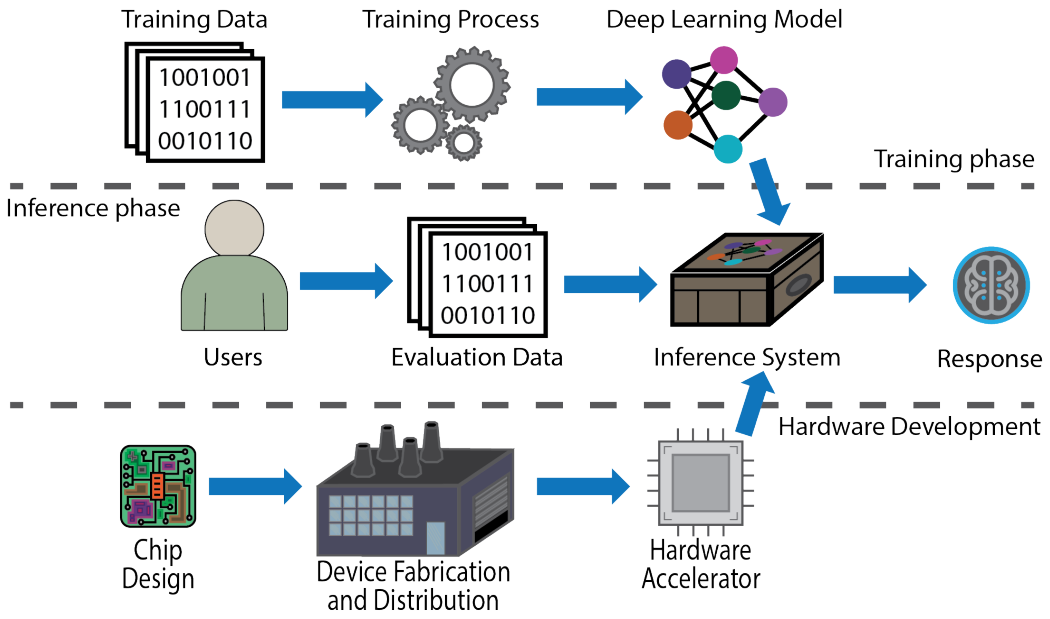

Figure 1.5: The consideration of hardware development in the deep learning deployment pipeline defines a phase of the process that is orthogonal to both the training and inference phases of the deep learning systems. However, this phase still has implications for the functionality of the system deployed.

1.4 Novel Perspectives in Deep Learning Security

As the hardware domain continues to become increasingly more integrated with deep learning systems, how adversaries can manipulate the critical failure modes of deep learning systems grows increasingly relevant. However, currently, the research exploring this connection is currently limited. The work contained in this dissertation establishes multiple pioneering works in bridging this gap and creating a foundation for future work in adversarial deep learning with a hardware perspective. Specifically, these contributions to the field of adversarial deep learning from the hardware perspective can be summarized as:

- Adversaries can inject backdoors into deep learning systems, typically through modifications to their model parameters, which allow adversaries to take control of a deep learning system through the use of a key input [61]. Attacks in the hardware domain and other low-level implementations don't typically have access to model parameters; we extended this body of work to compromise deep learning models through modifications to the mathematical operations of a deep neural network. This novel attack demonstrates the possibility of an adversary to backdoor deep learning systems through firmware, deep learning frameworks, hardware, or

other low-level implementations.

- Hardware Trojans are a serious concern in the hardware domain [12]. Hardware Trojans are malicious modifications to hardware designs that are potentially a widespread issue due to the ease of their injection, significant motivating incentives, and difficulty in detecting. We extend the study of hardware Trojans to the field of deep learning by developing an algorithm for embedding hardware Trojans in deep learning accelerator designs.

- Another prominent issue that is difficult to deter and detect in the hardware domain is piracy [85]. Watermarking is a popular method for mitigating piracy, which allows rightful owners to detect piracy upon fraudulent usage. We have made steps to extend this protection to deep learning hardware designs through a software-hardware co-optimized framework for embedding watermarks in Deep learning hardware.

- Finally, we explore the possibility of utilizing traditional deep learning attacks against deep learning systems in hardware-constrained setting. Such systems utilize well-optimized hardware and software, often without considering the security implication of the system. We demonstrate that in such situations, the deep learning systems remain vulnerable and, if utilized as a security solution, could introduce vulnerabilities that could compromise the system.

Chapter 2

Deep Learning Backdoors through Modifications to Model Operations

This work presented in this chapter was published in Global Conference on Signal and Information Processing (GlobalSIP) 2018.

2.1 A Novel Perspective on Deep Learning Backdoors

Recently, deep neural networks have become nearly synonymous with machine learning due to their capability of accomplishing notoriously difficult tasks with significant success. However, the internal reasoning of these models is obscured, making it difficult to defend them in adversarial settings. Their defense is further complicated by the high complexity of the models. Despite a neural network often achieving or even surpassing human-level accuracy in a task, they can be made to produce any adversarial responses with only minor modifications to a correctly evaluated input [48]. In the best scenarios, a human observer cannot even distinguish the difference between the adversarial example and the original. This characteristic has been exploited by a large number of studies to mount adversarial attacks on neural networks. Despite a multitude of proposed defenses, a solution for making neural network models truly robust to this vulnerability has been elusive.

In addition to this inherent vulnerability to a neural network's inferences, adversaries also have the ability to inject malicious behavior into neural networks to gain control of the system during

the inference phase [25, 104, 101]. Such attacks are called backdoor injection attacks. To this end, backdoor attacks have been developed to compromise a neural network through the modification of its weights so that the network behaves maliciously when a specific input key is presented to it. In fact, these backdoors can be inserted into the network in different ways, such as through poisoning attacks during the training phase [25] or through the injection of backdoors into a well-trained model [104, 101]. The presence of these vulnerabilities is detrimental to the use of deep neural networks in security-critical systems.

The benefit of backdoor attacks over adversarial example attacks is that the attacks tend to be more easily implemented in the inference phase. While adversarial examples are often generated at inference in response to the input/output behavior of a model, the input keys which activate a backdoor can be determined before deployment and so can be more easily generated after the initial overhead of injecting the backdoor functionality. Further, adversarial examples are generated with minor perturbations to a correctly processed input. As such minor perturbations can also reverse the attack making the adversary, require very fine-grained control over inputs. However, backdoors are generally activated by distinct triggers such as specific pixel patterns or even a specific physical object in an image. As such, backdoors typically require a smaller degree of control over individual elements of an input, making them more practical to conduct during deployment.

The major limitation of backdoor attacks is that they require access to the model pre-inference. Either the adversary needs to access the training phase of the model or to modify the model before its usage in the inference phase. Such access to deep learning models is heavily regulated, so it can be difficult for adversaries to gain the access necessary to embed the backdoor functionality. This typically requires a degree of physical subversion which puts the adversary at risk. Further, there are a number of trivial checks which can traditionally catch such attacks. For example, developers often keep backups of deep learning models, a trivial comparison between the backup and deployed models can verify if an adversary has manipulated a model during the inference phase. While such defensive measures are not fool-proof adversaries must approach backdoor injection intelligently to produce meaningful attacks.

The work presented here developed a novel algorithm for injecting backdoors into well-trained neural network models through the modification of the neural network's operations instead of altering the network weights. This could be used, for example, to mount attacks on firmware, deep learning frameworks, or low-level implementations of machine learning algorithms. This novel

perspective gives adversaries an avenue to backdoor injection that is completely orthogonal to all previous backdoor injection methodologies opening the door to completely new attacks that do not currently have any defensive measures in place to defend against.

The major contribution of this work is to demonstrate that the software domain and the infrastructure computing its operations can be algorithmically linked such that modifications to the infrastructure can directly control the deep learning model. We show that the proposed computation-level backdoor attacks could achieve very high success rates while only a very small subset of neuron operations need to be modified. There are various adversarial benefits that come from the possibility of embedding backdoors through such methodologies. For example, given that the mechanism of the proposed approach is distinct from previous methods, it may also be to integrate the proposed approach with other backdoor attacks. This would enable selectively targeted backdoor attacks, which only arise once the backdoor is injected from both the computational and software perspectives.

2.1.1 Contrasting the Adversarial Example and Backdoor Perspectives

As discussed in Section 1.2.1.2, backdoor injection can be summarized by the optimization problem seen in Equation 1.5:

$$\min_{\phi} \sum_{\mathbf{x}_t, \mathbf{y}_t \in \mathcal{D}_T} \mathcal{L}(F(\phi, \mathbf{x}_t), \mathbf{y}_t) + C \sum_{\mathbf{x}'_t, \hat{\mathbf{y}} \in \mathcal{D}'_T} \mathcal{L}(F(\phi, \mathbf{x}'_t), \hat{\mathbf{y}}). \tag{2.1}$$

This problem contains the dual benign and backdoor objectives of the deep learning scenario and attempts to simultaneously fulfill both. As such, models generated under this objective maintain some adherence to the benign task and only deviate from it under the specific backdoor trigger that activates the backdoor functionality. However, this backdoor perspective assumes that the target of manipulation is the deep learning model; which, due to the problem of catastrophic forgetting, requires this dual objective.

While closely aligned with the concept of backdoor injection, the implementation of the computational backdoor actually becomes more aligned with the understanding of adversarial examples. This is due to the nature of how computations change operations. While software-based backdoors often rely on altering the weights, are architectures of a deep learning model computational backdoors appear more like introducing perturbations to a computation. The traditional perspectives often appear as introducing a novel objective to the deep learning model's original training algo-

rithm. Altering the computational infrastructure of deep learning systems, however, appears more like introducing perturbations to a model and using a constrained optimizing algorithm to find the minimal perturbation to achieve the target behavior.

We can describe the generation algorithms as discussed in Section 1.2.2.1. To better align the generation algorithms with the problem of computational backdoors, we reformulate the unified optimization problem articulated in 1.6 as:

$$\begin{aligned} \underset{\mathbf{p}}{min} \quad & \mathcal{L}(F(\mathbf{x}+\mathbf{p}), \mathbf{o}_t) \\ s.t. \quad & d(\mathbf{p}) < \epsilon. \end{aligned} \tag{2.2}$$

Where $\mathcal{L}(\cdot,\cdot)$ is a loss function connecting the model's behavior given a modified input, $\mathbf{x}+\mathbf{p}$, and a desired target output, $\mathbf{o}_t$. $d(\cdot)$ measures the magnitude of the perturbation, $\mathbf{p}$, with respect to some criteria specific to the targeted scenario. From the perspective of perturbations on an input, especially images, there is a wide range of useful measures to be used as $d(\cdot)$. However, as we shift into the perspective of embedding modifications into a computational infrastructure, the type of perturbation introduced is distinct, directing the use of specific measures here.

Similar to the deep learning training process, the most powerful solutions to the adversarial example generation algorithms tend to be iterative approaches. While there is a wide variety of these approaches, generally they can be summarized as follows:

0) Initialize the algorithm with $\mathbf{p} = \mathbf{0}$.

1) Find a change in $\mathbf{p}$ which decreases the loss, $\mathcal{L}(F(\mathbf{x}+\mathbf{p}), \mathbf{o}_t)$.

2) Project the perturbed $\mathbf{x}+\mathbf{p}$ back into $d(\mathbf{x}+\mathbf{p}) < \epsilon$ if necessary.

3) If $\mathbf{x}+\mathbf{p}$ is an invalid input, project it back into the domain of valid inputs.

3) Repeat 1-3 until $\mathcal{L}(F(\mathbf{x}+\mathbf{p}), \mathbf{o}_t) = 0$, i.e. the adversarial objective is achieved.

One major thing to note in the adversarial example generation process is that unlike the injection of backdoors, there is no constraint to preserve the initial functionality of the deep learning model. This is because the perturbation is exclusively tied to the input, and so does not alter the model's behavior. However, as we transition into the backdoor perspective preserving this functionality becomes critical. As such, solving Equation 2.2 would not typically be considered a viable method

of injecting a backdoor when perturbing model weights or other algorithmic components. However, the perturbations to computational units are fundamentally different, allowing us to consider the use of similar algorithms in the domain of backdoor injection.

2.2 Proposed Backdoor Injection Methodology

2.2.1 Adversarial Setting and threat model

The threat model considered in this work assumes a white box which is consistent with many of the early works in backdoor injection. It considers a fairly weak defender where the adversary has been able to gain access to the deep learning model and its internals running on a system. However, we extend this threat model with the understanding that the adversary also has access to the computational infrastructure of the deep learning system. In the real world, such vulnerability often exists in through an untrusted supply chain or third-party IP integration. The attacker's goal in this threat model is to modify the system before it is deployed to the target application but has no control over the architecture or weights of the deep learning model.

As discussed in Section 1.2.1.2, several backdoor attacks have already been developed in the literature that attempts to alter the network's weights either directly or indirectly (i.e., through data poisoning attacks). However, a well-trained model consists of not only the weights but also the computing operations (e.g., multiply-addition, activation functions). Therefore, it is also possible to inject malicious functionality into the network operations. This work develops a method to modify these computing operations to produce the desired results for a very small selected set of input keys while the majority of outputs remain unchanged. In addition, we use malicious input keys that are close to the legitimate inputs to avoid detection during standard tests or applications. To simplify this process in this work, we consider a scenario in which all the operations in a deep neural network can be independently manipulated or modified. This requirement limits the usage of the attack for some scenarios but provides a foundation for further research.

For this work, we consider the image classification scenario. Within this setting, we consider two objectives for the adversary in this work. Namely, to produce targeted and untargeted misclassifications. In an untargeted scenario, the adversary is not seeking to produce a specific misclassification. Instead, any misclassification is sufficient for his objective. In this scenario, the backdoor injected should simply alter the model's predictions of the backdoor's key inputs to any

class that it does not come from. In the targeted scenario, the adversary intends to produce a specific output prediction. As such, the key inputs should produce a specific output prediction.

A major contributor to the development of this threat model is the software-hardware co-design development environments that have been growing in popularity in deep learning. These environments have largely grown due the power of deep learning but an inability to deploy them to many high-profile settings such as IoT and embedded computing. In these scenarios, the strictly constrained resources make it difficult to deploy the same high-powered systems used in other application domains. As such, software-hardware co-designs arise, which simultaneously optimize the hardware and deep learning model to work together, decreasing the resource utilization drastically over traditional general-purpose settings. In such settings, the hardware platform is not intended for general usage but for the usage of a specific deep learning model. An important direction for future research is the exploration of different threat models to generalize this family of attacks.

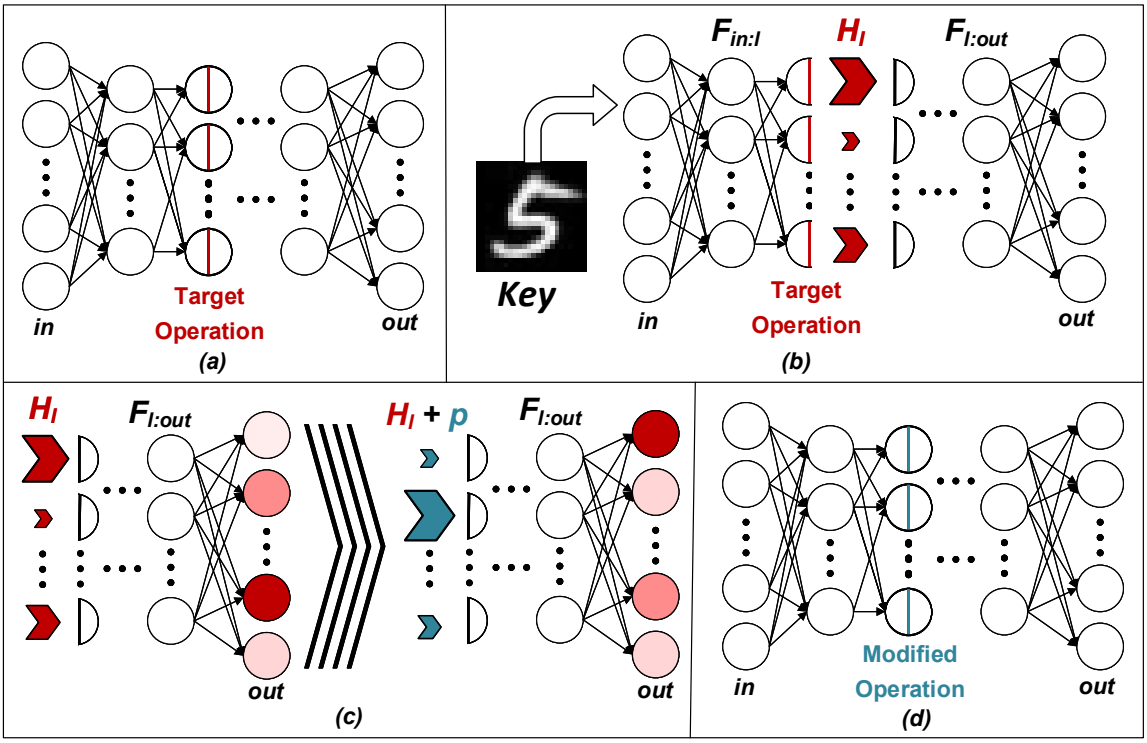

Figure 2.1: An overview of the proposed backdoor injection methodology. (a) The attack begins with an adversary targeting an operation in an arbitrary layer of a well-trained neural network. (b) The network is divided into sub-networks around the targeted operation. (c) The adversary calculates the required perturbation for the computing operation that alters the output classification to the desired one by using the proposed algorithm. (d) The backdoor-injected neural network.

2.2.2 Injecting Deep Learning Operation-Based Backdoors

In this section, we present the detailed steps of the methodology for performing backdoor attacks on neural network operations. To better illustrate the proposed methodology, we begin by assuming that the adversary has isolated a targeted layer of operations that will be executed by a single set of computational blocks in the computational framework. For example, this could be a set of operations isolated to a known array of functional blocks in a hardware design. The adversary has the flexibility to arbitrarily choose the targeted operations, the modified predictions, and the input key used to unlock the backdoor. The perspective proposed could be easily extended to a multi-layered attack, as discussed in Section 2.2.2.3.

The proposed methodology first establishes a perspective that allows the adversary to isolate the target operations by considering the deep learning model as two sub-networks centered around the target operations. Then, the adversary is able to characterize the benign response of the target operations by feeding target key samples into the model and evaluating the outgoing and incoming latent representations of the sub-networks. The adversary algorithmically incorporates the computational modifications into the model as a perturbation on the input of the second sub-network. Using a modified adversarial example generation algorithm, the adversary can then determine the perturbation required to embed the target backdoor functionality. Finally, the adversary utilizes the latent representations and perturbation found to modify the computational framework to generate the backdoor. An illustration of the process is presented in Figure 2.1.

2.2.2.1 Decomposing the Target Model into Sub-Networks

The process of isolating the sub-networks is facilitated by the network's layered structure. Each layer, l, in the network is defined by a set of weights, W_l, which is multiplied by the layer's input vector, H_{l-1}, before applying an activation function. For example, the function implemented by a fully-connected layer can be expressed as

$$H_l = \sigma_l(W_l \cdot H_{l-1}), \tag{2.3}$$

where σ_l represents the layer's activation function. Within this structure, we can easily define an order of layered operations used to produce the layer. In other words, to compute the layer first, a sequence of multiplies will occur, then a sequence of additions, and the a sequence of applying

the activation function. This layered structure of operations largely defines all deep learning model layers as it enables the ability to parallelize these models and enables their efficient computation. Further, by feeding the output of each layer to the input of the subsequent layer, the entire network can be characterized as a sequence of layered computations.

We can then define sub-networks of the neural network as sequential collections of these layered operations. Defining these sub-networks mathematically becomes mathematically complex due to the infinite variability of possible model layers. To simplify and generalize this framework, we will constrain our ability to define sub-networks by only allowing them to be divided around the activation functions of the model. However, we note that this choice is arbitrary and can be trivially extended to any operation in the model. We use the notation $F(x)_{i:j}$ to represent a sub-network where i and j represent the beginning and end layers of the sub-network, respectively. The adversary commences the attack by selecting a desired targeted operation and determining the layer, l, containing the target operations, as shown in Figure 2.1(a).

We then divided the neural network into two sub-networks around the targeted operation such that:

$$F(\mathbf{x}) = F_{l:out}(\sigma_l(F_{in:l}(\mathbf{x}))) \tag{2.4}$$

holds true. We use in and out to denote the input and output layers, respectively. For example, we consider a simple two-layer network: $F(x) = \sigma_2(W_2\sigma_1(W_1\mathbf{x}))$. A backdoor attacker targeting the activation function of this first layer would isolate the sub-networks $F_{in:1} = W_1\mathbf{x}$ and $F_{1:out} = \sigma_2(W_2F_{in:1})$. Likewise, larger n-layered models can be broken into two sub-networks around any of its n activation functions allowing the adversary to isolate that set of parallel operations. This defines the two sub-networks, $F_{in:l}(\cdot)$ and $F_{l:out}(\cdot)$, and target operation, $\sigma_l(\cdot)$, that compose the target neural network.

2.2.2.2 Characterizing the Target Operations

Once the adversary has isolated the target operation by decomposing the mode into two sub-networks, the adversary can use these sub-networks to characterize the operation of the target operational layer using the first sub-network, $F_{in:l}(\cdot)$. This is done by feeding the target key inputs into the sub-network. By doing so we can define the model's latent representations before and after the target operation. We can mathematically define finding these latent representations using the

following:

$$H_{l-1} = F_{in:l}(\mathbf{x}_k) \tag{2.5}$$

and

$$H_l = \sigma_l(H_{l-1}) \tag{2.6}$$

for $\mathbf{x}_k \in \mathcal{X}_k$ where $\mathcal{X}_k$ is a set of key samples.

These latent representations represent the values that will be seen entering and exiting the computation's implementation when the model is executing the key samples. Note that models with high parallelization imply that these latent spaces will be very high dimensional. Similarly, targeting smaller operational layers implies smaller latent representations. Considering this factor becomes important depending on the target scenario. For example, if a limited number of computational blocks are being re-used to compute multiple operations, targeting layers with high dimensionality implies modifications to a single computational will affect multiple operations. Such factors can have a large effect on the backdoor in practice and must be considered.

2.2.2.3 Perturbing the Targeted Operations

We then introduce an algorithmic perturbation to the model that simulates modifications to the target operations by introducing a perturbation to the input of the second sub-network. This can be understood mathematically as redefining the model with an internal perturbation, $\mathbf{p}$.

$$F'(\mathbf{x}) = F_{l:out}(\mathbf{p} + \sigma_l(F_{in:l}(\mathbf{x}))) \tag{2.7}$$

With this understanding, we observe that finding a perturbation that produces the desired backdoor functionality can be considered synonymous with finding a perturbation on the input of the second sub-network when its input is H_l. In other words, we wish to solve the optimization problem:

$$\begin{aligned} \min_{\mathbf{p}} \quad & \mathcal{L}(F_{l:out}(\mathbf{p} + H_l), \mathbf{o}_t) \\ s.t. \quad & d(\mathbf{p}) < \epsilon, \end{aligned} \tag{2.8}$$

While from the adversarial example perspective, the perturbation, $\mathbf{p}$, would be absorbed into the pixels of the input sample, here $\mathbf{p}$ is injected through the alteration of computational entities. This fundamental difference in the method used to inject the perturbations has significant

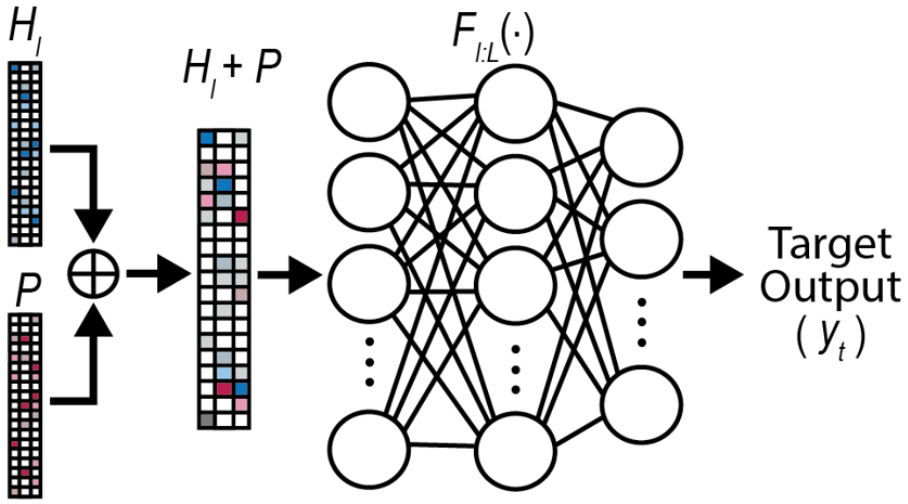

Figure 2.2: Dividing the model into sub-networks enables us to modify adversarial example generation algorithms for determining the perturbations require on internal layers.

implications for the resultant behavior of the backdoor. In the adversarial example attack, the perturbation only affects a single image. However, in the case of a modifying computational entity, all operations which touch the computational entity are subject to any modifications to it.

As such, it is important when modifying the computational entities to produce perturbations with a minimized effect on the untargeted operations. We have some control over this property from the algorithmic domain with the constraint $d(\mathbf{p}) < \epsilon$. When defining the optimization problem for a specific application domain, this constraint should be well aligned with the type of modification to produce a perturbation that is minimized in a way that benefits the type of modifications embedded.

To illustrate, when embedding modifications into hardware blocks, any modification will introduce overhead to the design. Further, logic modifications can be naturally designed to generate a piece-wise change in the target operations. Such modifications are designed to have a large change on localized regions of an input. In order to minimize the functional impact of such modifications on the hardware, it is important to ensure that very few computation blocks in the hardware are modified and that each computational block only alters a small number of operations. In such a setting, selecting a $d(\cdot)$ that constrains the number of operations to be modified can significantly decrease the overhead and functional impact of the embedded modifications on the hardware.

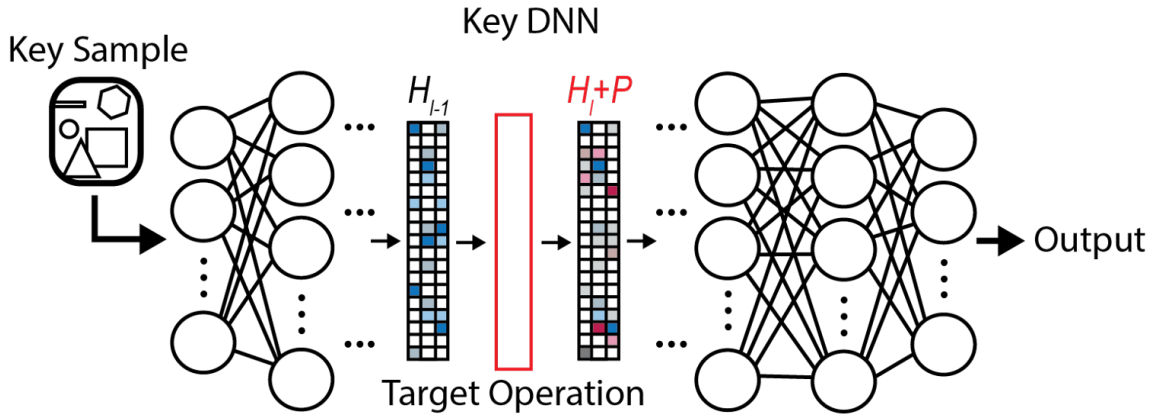

Figure 2.3: Once the backdoor perturbation, **p**, is determined, we rejoin the sub-networks while incorporating the pertturbation in the target operation embedding the backdoor functionality in the deep learning model.

2.2.2.4 Algorithmically Determining the Backdoor Perturbations

Next, we can extend the concept of adversarial example generation to the generation of operational perturbations. To generate sparse perturbations, we propose a novel approach to determine the modifications needed to insert the backdoor, which is a modified version of the JSMA. As shown in Figure 2.1(b), the first sub-network, $F_{in:l}$, generates an internal activation H_l at the target operation for the selected input key $\tilde{\mathbf{x}}$, which is given by:

$$H_l = F_{in:l}(\tilde{\mathbf{x}}). \tag{2.9}$$

We form a Jacobian Matrix, J, by calculating the gradient of output classification, c, from the set of possible classifications, C, with respect to the targeted operations, as expressed in Equation (2.10).

$$J(H_l) = \left[\frac{\delta F_{l:out}(H_l)[c]}{\delta H_l}\right]_{c \in C}. \tag{2.10}$$

Then we build a Saliency map, S, by applying a set of rules to the Jacobian matrix, which indicates how the targeted operation should change in order to generate the modified output. We customize these rules to fit the specific goals of the adversary. For example, a backdoor attack that seeks to

classify the selected input key to a targeted class could use Equation (2.11),

$$S(\mathbf{x}) = \left|d\mathbf{x}_p^-\right| + d\mathbf{x}_t^+ + \sum_{i \neq t, c_0} \left|d\mathbf{x}_i^-\right|, \tag{2.11}$$

while Equation (2.12) can be utilized for the untargeted scenario.

$$S(\mathbf{x}) = \left|d\mathbf{x}_p^-\right| + \sum_{i \neq c_0} d\mathbf{x}_i^+. \tag{2.12}$$

The $d\mathbf{x}$ values refer to the columns in the Jacobian matrix linking the targeted operations to the predicted, $d\mathbf{x}_{c_0}$, or targeted, $d\mathbf{x}_t$, classes as well as other possible predictions. The i_{th} entry in the saliency maps indicate a positive or negative correlation between modifications to the i_{th} input and the specific adversarial goal used in generating the customized saliency map. We iteratively calculate the saliency map and alter the network operations highlighted by the most extreme of these values to determine the perturbation, $\mathbf{p}$, that produces the malicious behavior, as shown in Figure 4.4. The two sub-networks are recompiled with the altered operation, illustrated in Figure 2.3.

2.3 Experimental Evaluations

2.3.1 Experimental Procedures

2.3.1.1 Datasets and Neural Network Models

We use MNIST and CIFAR10 datasets to evaluate the proposed methodology. Both datasets follow the image classification task and are composed of 10 mutually exclusive classes. The MNIST dataset contains $60,000$ training images of hand-drawn digits correctly labeled with their respective digit class and an additional $10,000$ in testing examples. Each image is a 28×28 single-channel image. The CIFAR10 dataset also contains $60,000$ training images of low-quality real-world subjects correctly labeled with their respective class and an additional $10,000$ in testing examples. Each image is a 32×32 three-channel RGB image. These datasets constitute low-dimensional datasets that are widely used for benchmarking deep learning systems and prototypes.

To classify these datasets, we utilize two 7-layered CNNs. The detailed architectures of the models implemented in our experiments are summarized in Table 2.1. We pre-train the networks to achieve test accuracies above 99% and 80% on the MNIST and the CIFAR10 datasets, respectively.

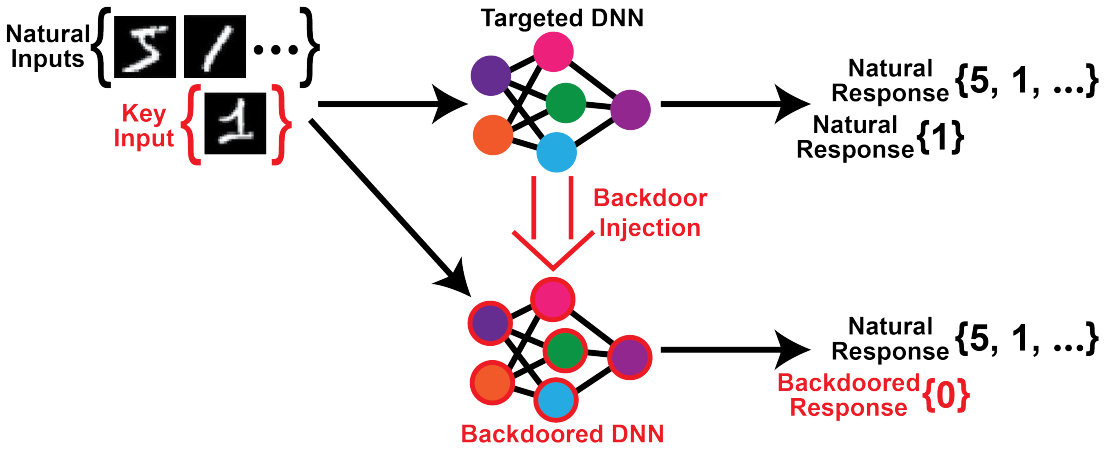

Figure 2.4: In these experiments, we attempt to embed a backdoor into the deep learning classifiers through simulated perturbations on the model operations. While the original model exhibits a specific functionality when computing the testing inputs after embedding the backdoor we are able to successfully alter the functionality of the model on a target key input while maintaining this prior behavior.

These accuracies are consistent with other small-scale models used in similar scenarios. We then consider these models as the original benign models to inject operational modifications into using the proposed algorithm. We run the experiments on a high-performance cluster node with NVIDIA Tesla GPUs. The deep learning framework is implemented using the Tensorflow package [1].

For these experiments, we subdivide the Tensorflow models into two sub-networks layer-by-layer. We then introduce an external perturbation to the input of the second sub-network. We determine a minimal perturbation required to inject the backdoor while enforcing a sparsity constraint on the perturbation. We do not use a tight constraint, however, but instead allow the algorithm to progress for 50 iterations of the algorithm, which ensures $|\mathbf{p}|_0 < 50$. The two sub-networks are then reconnected with the novel internal perturbation applied when the target latent representation is observed, and the model is evaluated for its performance.

To simulate the scenario in which the adversary crafts an input key closely resembling legitimate inputs, we pick a single test input as the malicious input key and remove it from the testing data. We then target the activation functions of each hidden layer individually, performing both targeted and untargeted attacks. The final, 7_{th}, layer is excluded due to its proximity to the primary output. We repeat each experiment for at least 1000 randomly selected input keys from the test data for both the MNIST and the CIFAR10 classifiers.

	MNIST		CIFAR-10	
layer	type	# neurons	type	# neurons
1	conv 20	15680	conv 32	28800
2	conv/max 40	31360	conv/max 64	50176
3	conv 60	11760	conv/max 128	18432
4	conv/max 80	15680	conv/max 128	2048
5	conv 120	5880	dense	1024
6	dense	150	dense	180
7	dense	10	dense	10

max-pooling size: 2x2, kernel size: 3x3

Table 2.1: Summaries of network architectures

2.3.1.2 Experimental Results

We evaluate the effectiveness of the proposed methodology by first examining the success rate (i.e., the percentage of key samples which successfully generate the malicious behavior). Our experimental results demonstrate that our algorithm generally achieves a very high success rate. In particular, all untargeted attacks on both classifiers achieve success rates of 100%. Further, the targeted attacks yield an average success rate of over 99%, with the lowest average success rate for attacks on a single layer being above 96%. The success rate of adversarial examples is frequently understood to be linked with the tightness of the similarity constraint. As such, these results could be easily improved by allowing the algorithm to progress for more than 50 iterations at the cost of increasing the modifications required to produce the backdoor injection.

We now examine the number of modifications needed to produce this malicious behavior. The performances while targeting each layer of the network is recorded individually and presented in Figure 2.5 and Figure 2.6 for MNIST and CIFAR10, respectively. As we observe from the graphs, the modifications required by the backdoor injection remain very small, $< 1\%$ of the layer's operations being modified, for all targeted layers of the model except the 6_{th}. Further, it can also be seen from Figure 2.5 and Figure 2.6 that the untargeted attacks require less modifications than the targeted counterparts, which is consistent with our expectation as the untargeted attacks are more relax compared to targeted attacks.

This analysis closely correlates to the impact of the attack in terms of how significant the modifications will be on the computational infrastructure. For example, if embedding hardware modifications in a deep learning accelerator, each modification requires some additional overhead and increases the likelihood that the modifications could be discovered while analyzing the hardware. As

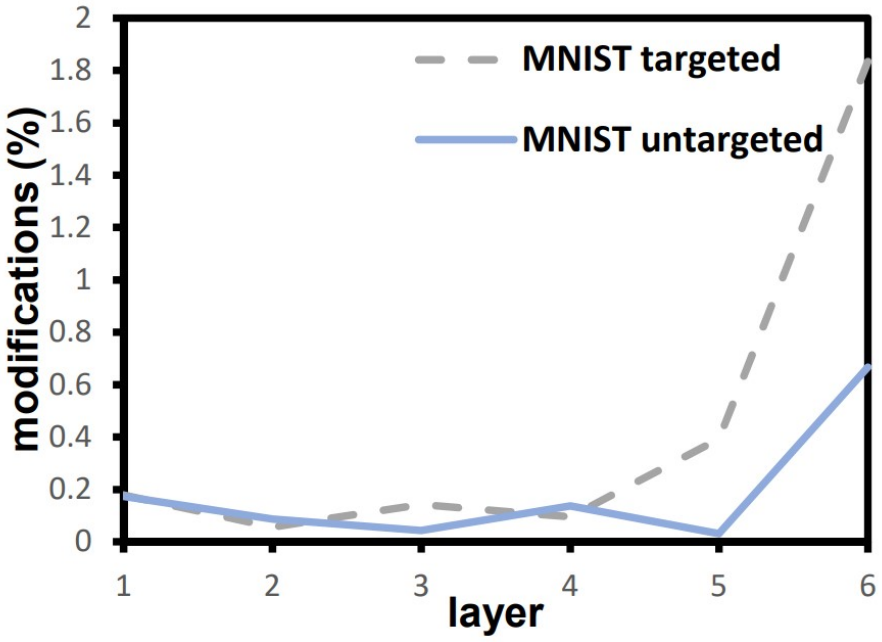

Figure 2.5: The average modifications of neurons needed in an MNIST classifier per targeted layer.

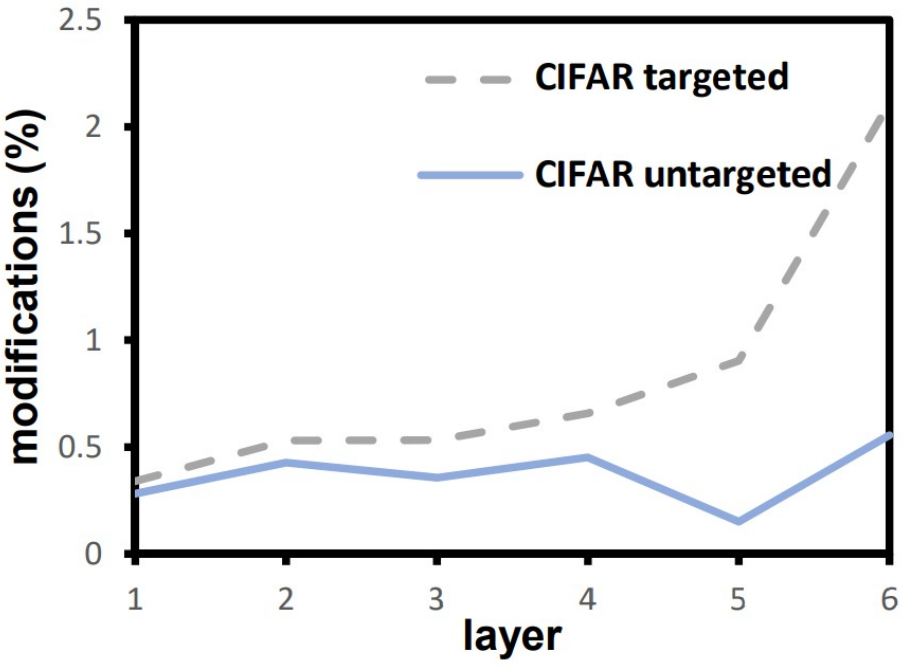

Figure 2.6: The average modifications of neurons needed in a CIFAR10 classifier per targeted layer.

such, this impact also indicates the level of stealthiness of the backdoor from inspection techniques. It should be noted here that in some scenarios, the percentage of modifications is less important than the overall magnitude of the modifications. In such cases, attacks targeted at the latter layers of the model may be preferable despite the significant increase in the percentage of required modifications, as the total number of modifications may be smaller.

Finally, we evaluate the functional impact of the injected backdoor on the remaining testing dataset after the removal of the backdoor key samples. For this evaluation, we simply verify the modified model's prediction on the testing dataset and compare it with the predictions of the original model. This allows us to determine what percentage of the model's testing data has been altered by the einjected perturbation. In our experiments, the classifications produced by the modified model are unaltered from those of the original. In other words, the backdoor-injected models would behave absolutely normally if the input keys were not present. As such, we conclude that in this setting, the proposed methodology yields an effective yet stealthy backdoor attack. We note however that this is largely due to the specificity of the activation method used in activating the perturbations.

In practice, limitations of the modifications embedded in the computational infrastructure would be a significant contributor to this property. In this evaluation, we considered an ideal setting in which the modifications have a very reliable activation method and only activate under the correct latent representation. In practice, the triggering mechanism of the modification would likely have a less accurate trigger and be subject activation when similar but not exact latent representation. When transitioning to attacks on different computational infrastructures, the sensitivity of the modifications activation mechanisms should be considered and designed to minimize false activation of the perturbations.

2.4 Conclusions

2.4.1 Summary

The field of adversarial deep learning has revealed a number of security concerns for deep learning systems. These include inherent vulnerabilities which allow an adversary to search the models input space to generate adversarial examples. Besides such inherent vulnerabilities in a neural network's inferences, an adversary also has the ability to inject malicious behavior into neural networks to gain control of the system between the training and inference phases. To this end,

backdoor attacks have been developed to compromise a neural network through the modification of its weights so that the network behaves maliciously when a specific input key is presented to it. In fact, backdoors can be inserted into the network in different ways, such as through poisoning attacks during the training phase or through the injection of operation modifications into a well-trained model. The presence of these vulnerabilities is detrimental to the use of deep neural networks in security-critical systems.

In this work, we developed a novel algorithm for injecting backdoors into well-trained neural network models through the modification of the computational infrastructure of a model instead of altering the network weights. This could be used, for example, to mount attacks on firmware or low-level implementations of machine learning algorithms. To the best of our knowledge, all previous backdoor injection methods rely on altering network parameters or training data from the algorithmic domain. Our attack is orthogonal to these attacks and thus would be extremely difficult to detect using defensive techniques which analyze network parameters.

We show that the proposed operation-level backdoor attacks could achieve very high success rates while only a very small subset of neurons need to be modified. Given that the mechanism of the proposed approach is distinct from previous methods, it may also be integrated with other backdoor attacks to further enhance the attacking effectiveness. Additionally, as this attack does not alter the network parameters, it would be extremely difficult to defend against using traditional defense techniques that detect changes in these operations. This operational backdoor lays the foundation for our future work in embedding modifications into hardware designs which target the behaviors of the deep learning model, allowing additional abnormal functionality to be introduce through the hardware.

Chapter 3

Compromising Deep Learning with Hardware Trojans

In this chapter we present, work that was published in International Symposium on Circuits and Systems (ISCAS) 2019. It was also selected for presentation in Defcon 2018 as an invited talk.

3.1 Neural Network Hardware Implementations and Trojans

Hardware designs are heavily reliant on the horizontal supply chain. This is primarily due to the significant cost reductions involved in outsourcing specific manufacturing processes. This reliance introduces blind sports to the supply chain as designers are often unable to view inside the manufacturing process and leaving much of the manufacturing process untrusted. Within these untrusted processes, adversaries have the ability to subvert the interests of the designer and modify a design for their own agenda. This issue is further compounded due to the fact that these horizontal supply chains are frequently globalized and governed by multiple regulatory and government entities. This makes it possible for competing governments, corporations, or individuals to apply pressure on actors within a supply chain to compromise hardware designs, even against their own desires. As such, hardware Trojan attacks, where an actor modifies a hardware design for malicious intent, are a major concern in the industry of hardware development.

Further, despite significant research interest in detecting and mitigating the potential of

adversaries embedding hardware Trojans in a design, the issue remains largely unsolved. This is largely due to the ability of hardware Trojans to hide within the noise generated by the random process variations of modern nanometer scale designs. Many reliable methods of Trojan detection utilize destructive techniques which compromise the device, which in the best-case scenario, can only detect a Trojan embedded in the destroyed device. This makes it costly to perform such an analysis without a guarantee that the remaining devices are not compromised. Indeed, the ease with which adversaries can inject hardware Trojans into a design through the modern supply chain, along with the cost and difficulty of detecting them in practice, make them a serious concern for modern industries.

As illustrated in Figure 3.1, this chapter explores to the possibility of embedding hardware Trojans in deep learning accelerators to manipulate the deep learning models executed on them. To the best of our knowledge, the work presented here is the first to directly target deep learning hardware accelerators using hardware Trojans.

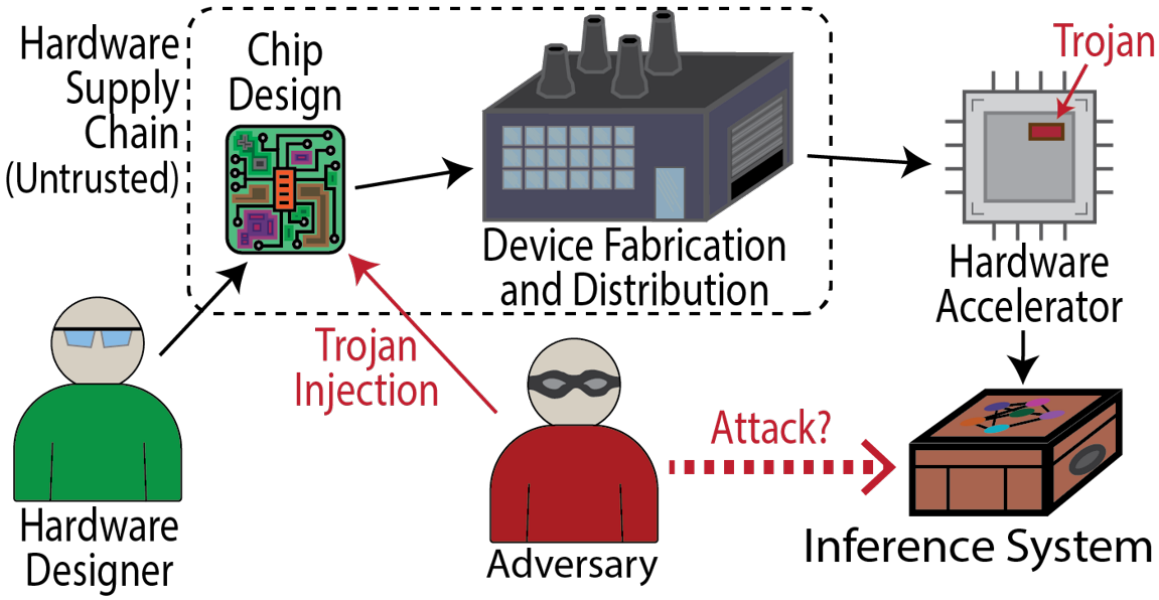

Figure 3.1: In the modern manufacturing industry, hardware supply chains frequently contain multiple untrusted processes where designers are unable to verify the security or reliability of their outputs. Adversaries can take advantage of these untrusted processes and inject modifications into hardware designs that introduce malicious functionality to the design. In this chapter, we explore the possibility of an adversary embedding such hardware Trojans into a deep learning hardware accelerator and compromising the models executed on that platform.

3.1.1 Hardware Design and Security

3.1.1.1 Neural Network Hardware Accelerators

As the field evolves, there has been an elevated desire recently in relocating the inference phase of machine learning algorithms to edge devices. To this end, hardware solutions that improve the efficiency of neural networks are necessary for deploying complex models in resource-constrained platforms, such as Internet-of-Thing (IoT) devices. In the literature, various hardware accelerator designs for neural networks have been proposed [170, 26, 154], most of which can be generally classified into either temporal or spatial architectures according to the parallelism pattern of the multiply-accumulators (MAC) [143], as seen in Figure 3.2. Since neural networks are becoming deeper and deeper, it is infeasible or extremely costly to implement fully-parallel architectures in practice. Therefore, MAC operations are often organized into a number of folding sets and executed in a time-multiplexed manner. Similarly, other basic computational blocks required by neural networks, such as pooling and activation functions, are also parallelized or time-multiplexed. In particular, recent work has proposed an optimized folding scheme to increase the power and area efficiency for VLSI implementations of neural networks [154].

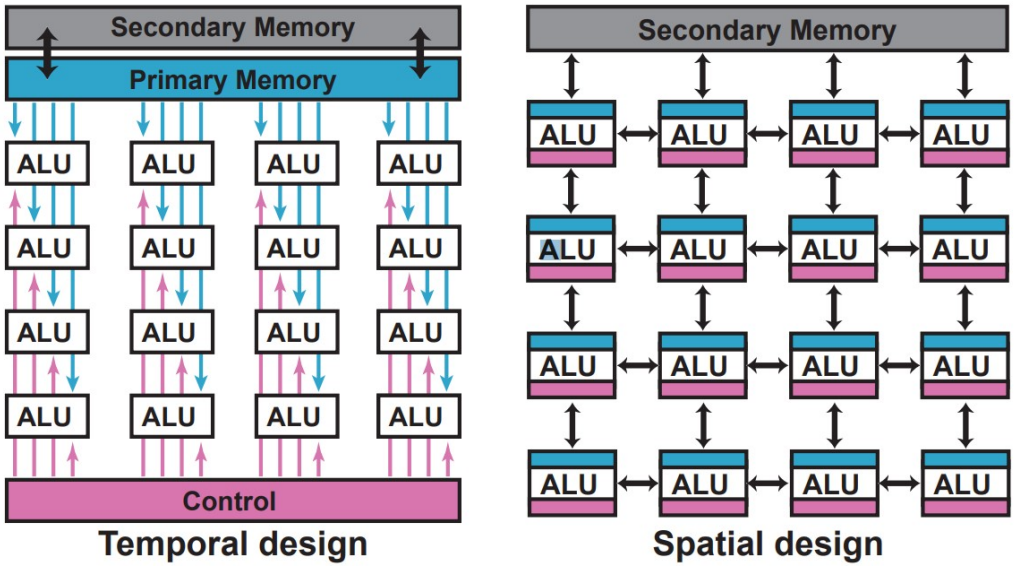

Figure 3.2: Two common parallelization paradigms [143].

More recently, the benefits of co-designing hardware and algorithmic components have been gaining recognition [89]. Quantized deep learning hardware has seen particular success, where hard-

ware noise is introduced through aggressive weight and activation quantization. However, it is possible to handle this noise from the algorithmic perspective by modeling and incorporating that noise into training protocols reducing the negative effects of hardware quantization [31, 130, 178]. Recently, the utilization of such algorithmic-hardware co-design techniques has been enabling deep learning on the edge and in IoT settings [174]. While typically deep learning capable IoT devices utilize deep learning components designed for general-purpose processors, in these resource-constrained settings require more aggressive optimization, including quantization, network pruning, and sparsification [153, 65]. As discussed with quantization, these techniques can be co-optimized with the hardware design to improve performance and efficiency past which the two can achieve independently [77].

3.1.1.2 Hardware Trojans and Backdoors

Hardware Trojans are critical threats for hardware devices through the globalized untrusted supply chain [131, 22, 103]. A hardware Trojan is mainly comprised of two components, i.e., a trigger and a payload. Specifically, the payload achieves the Trojan's malicious functionality, while the trigger activates the payload under certain rare conditions. Two examples of simple Trojan designs can be seen in Figure 3.3. In order to evade design-for-trust and other detection methodologies from detecting the injection of malicious hardware modifications, a successful hardware Trojan is required to be extremely tiny and stealthy [131]. Moreover, it becomes nearly impossible to detect the presence of Trojans in modern deep nanometer technologies, especially when a modification's effects fall within the range of manufacturing process variation. Additionally, the fact that the internal structures of neural networks are highly-complex and intractable further complicates hardware Trojan detection.

Various hardware Trojan designs for basic arithmetic operations have already been extensively studied [129, 54]. Thus, many of these existing Trojan designs could also be applied to the operations of neural networks. For example, Trojans can be injected into a multiplier by slightly altering the path delays of the circuit that only activates when both inputs exhibit specific transitions simultaneously [54]. As opposed to these basic arithmetic elements, this work proposes to study the hardware Trojan designs for unique operations in neural network implementations. We consider a threat model similar to the state-of-the-art in the field of hardware Trojans, in which an adversary is assumed to have access to the netlist of a neural network design through an untrusted supply chain [95, 49, 78].

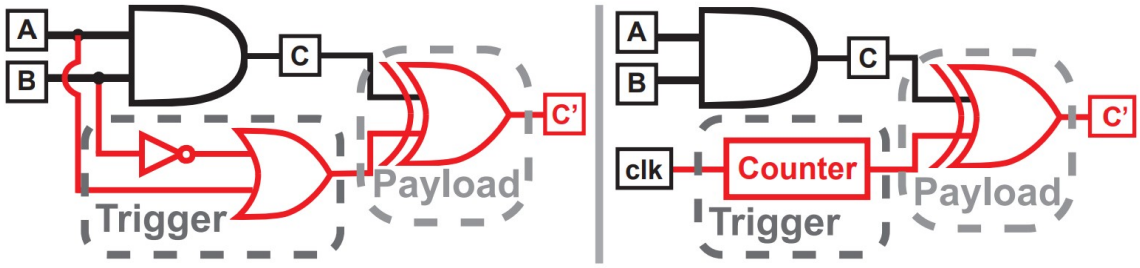

Figure 3.3: Simple hardware Trojan designs.

3.1.2 Software Trojans in Neural Networks

In the context of machine learning, the adversary could inject software Trojans into the model by maliciously altering its weights so that the neural network will malfunction when the Trojan is triggered. In the literature, several works on neural network software Trojan attacks have been developed [53, 104, 101]. From the supply chain perspective, maliciously intended modifications to these devices during the process can further provide attackers with new capabilities of altering the functions of internal neurons and causing adversarial functionality. Hardware Trojans can be inserted into a device during manufacturing by an untrusted semiconductor foundry or through the integration of an untrusted third-party IP. Furthermore, a foundry or even a designer may possibly be pressured by the government to maliciously manipulate the design for overseas products, which can then be weaponized. Therefore, it is of great importance to examine the implication of hardware Trojan on neural networks. In this work, we expand the attacks on neural networks from the training and application phases to the production phase. These attacks are typically considered synonymous with the deep learning backdoors discussed in Section 1.2.1.2 but have a broader range of adversarial objectives only one of which may be to embed a secret backdoor.

3.1.3 Defining Attacks on Deep Learning Hardware

In this section, we present the modern taxonomy of deep learning security and expand it to incorporate attacks on deep learning systems through hardware platforms. We further establish a general threat model for hardware-based attacks.

3.1.3.1 The Taxonomy of Deep Learning Security

A taxonomy is a tool used to categorize knowledge and concepts in a particular subject. They are useful in developing a broad understanding of a field of study. In the literature, the taxonomy of attacks on neural networks [10, 122, 123] is divided into four domains: the phase at which the attack is initiated, the goal of the attacker, the scope of the attack, and the attacker's knowledge of the system, as shown in Figure 3.4. In particular, the attacks are classified into two phases according to the stages of the neural network: the training phase and the inference phase [123]. An attack in the training phase seeks to take control of the training algorithm by maliciously altering the model training. On the other hand, an attack in the inference phase attempts to explore possible flaws in the well-trained model without tampering with the network. Attacks in the hardware domain are fundamentally different from both of these perspectives. Instead, we consider an attack on the computing infrastructure that, in general, is unchangeable during the inference phase. We call this the production phase, a stage in the design of deep learning systems where the infrastructure not directly related to the deep learning system is prepared. Note that fault injection attacks on the neural network [99] still fall into the inference phase since the described threat model assumes it occurs after full deployment.

The goals of the attacker can be organized into reliability or privacy attacks. Reliability attacks attempt to degrade the service provided by the deep learning system. Reliability attacks either degrade the integrity of the model (producing harmful outputs) or the availability of the model (making outputs less useful). Privacy attacks attempt to siphon some useful information from the model, such as the model parameters, training data, or user data. Attacks on deep learning systems can either be targeted when the adversary has a specific goal or untargeted when the adversary has a very general goal. Targeted attacks are typically considered stronger attacks because a targeted attack can always enable an untargeted attack by selecting an arbitrary goal. But untargeted attacks cannot always be successfully converted to a targeted attacks.

Another axis of classification is the level of knowledge the attack has. Whitebox attacks assume that an adversarial has complete access to the entire system, including its training and internal state. A black box attack, on the other hand, assumes that the adversary has very little knowledge of the system. Typically this scenario restricts the adversary to only considering input/output response pairs observed after deployment. While grey box settings indicate those in which the adver-

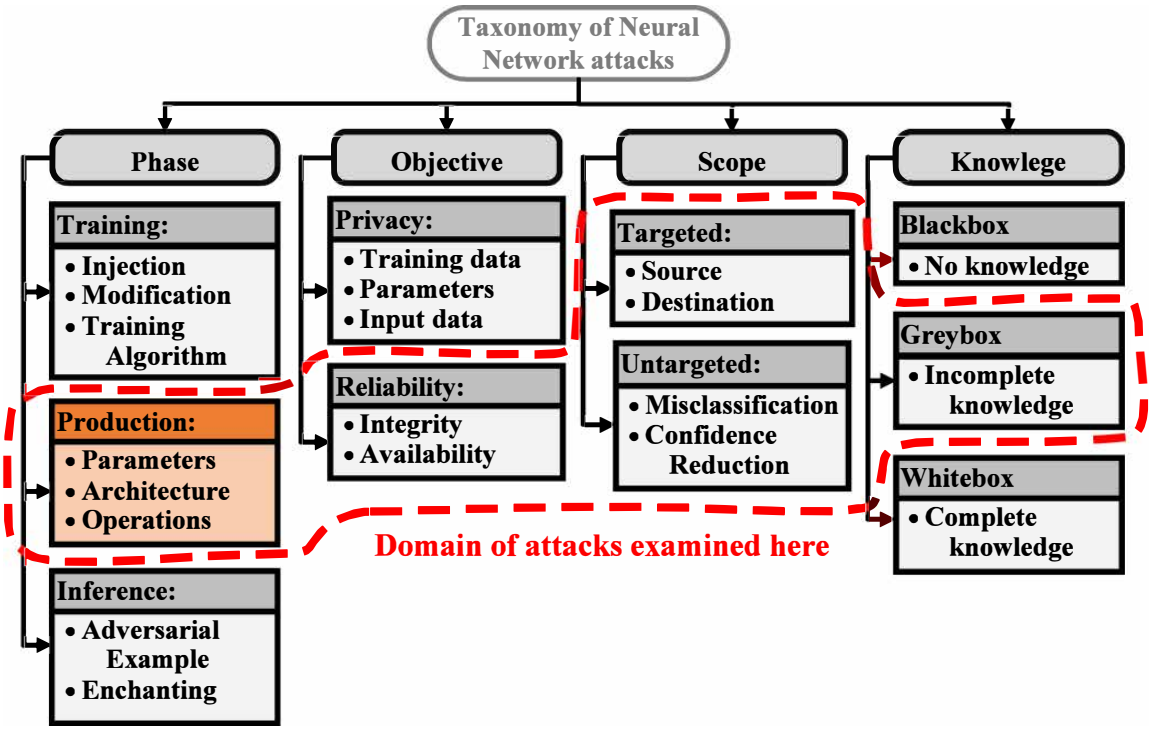

Figure 3.4: The expanded taxonomy of neural network attacks.

sary has some restricted access to some knowledge about the deep learning system but not all. This hardware Trojan attack is a grey box, for example, because it requires access to model parameters after training but does not require access to any of the training procedures or inputs. In sum, we consider the hardware Trojan attack during the production phase to compromise the reliability of neural networks with both targeted and untargeted scopes in this work, as circled in Figure 3.4.

3.1.3.2 General Threat Model of Attacks on Deep Learning Hardware

Unlike software Trojans, hardware Trojans consider the malicious modification of the original circuitry [22, 148]. An inserted hardware Trojan will change circuit functionality by adding, deleting or modifying the components to wrest control from the original chip owners. As opposed to software Trojans, hardware Trojans would have both capabilities of changing the weights and altering the functionalities of specific neurons depending on the threat model. This difference undoubtedly leads to vastly distinct insertion and design strategies for neural network hardware Trojans. Indeed, given that the hardware Trojan produces new threat models with no equivalent software counterpart,

attack, and defense scenarios must be first studied.

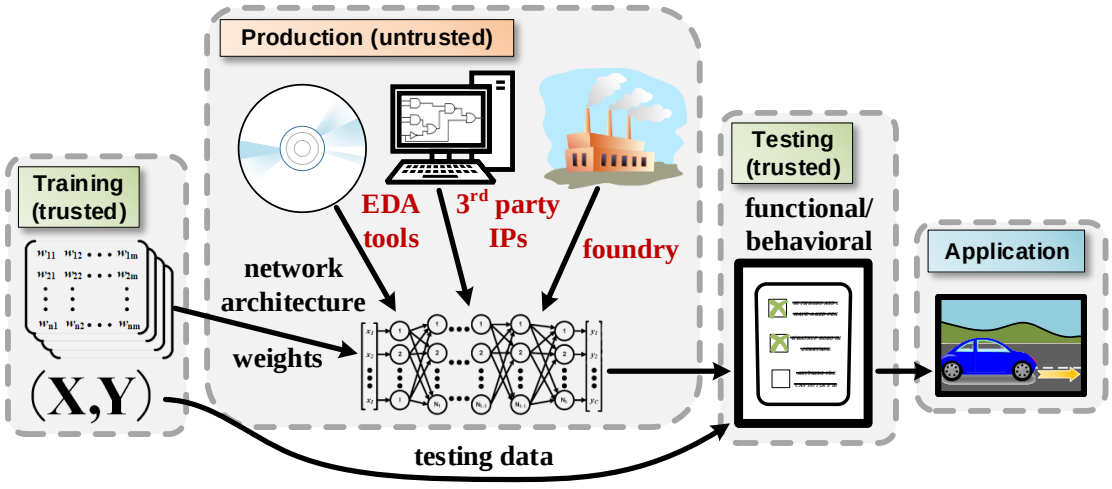

Figure 3.5: The adversarial setting considered in this work. The hardware perspective introduces a novel attack vector that has not been considered in prior works. This enables both traditional and novel attacks to be conducted in the hardware supply chain against deep learning systems.

In this work, we consider a threat model that assumes an adversary is positioned in the supply chain of an integrated circuit containing a well-trained neural network model, as shown in Figure 3.5. This threat model is particularly relevant given that many companies wish to use state-of-the-art offshore technologies to remain competitive in the market, especially for neural network devices whose performance is crucial for real-time applications. It is also plausible that, due to potential speed-ups and improvements in power consumption, the designer desires to hard-wire the network parameters. This setting would give the adversary direct knowledge of architecture and all weights associated with the well-trained model. However, the adversary would not have knowledge of the training or test data.

The objective of the adversary is to insert a hardware Trojan into the original design of the neural network circuit, forcing a specific trigger input to be misclassified to either a targeted or an untargeted class. Under this scenario, the adversary can modify both the weights and the functionalities of circuit components prior to shipment. In order to evade detection, the adversary should ensure the hardware Trojan is stealthy enough such that the predictions for the unknown test data are completely unmodified. In addition, the physical footprint of the hardware Trojan must remain sufficiently tiny; thus, the Trojan-injected circuit would be difficult to differentiate from the

original "golden circuit." In this work, we focus on the hardware Trojan attack on neural network circuit components, while we expect the hardware Trojan targeting the weights would yield a similar impact as the software Trojan or fault injection attack.

3.2 Injecting Hardware Trojans in Neural Networks

3.2.1 General Framework

The proposed framework consists of two main steps: (i) malicious behavior generation, i.e., determining the operation(s) to alter with the injected Trojans and the corresponding perturbations, and (ii) hardware Trojan implementation, i.e., designing the trigger and payload circuitry. Our proposed methodology provides an adversary the flexibility in selecting the targeted layer of a neural network for injecting hardware, Trojan. Without loss of generality, we assume the targeted layer is layer l. An example of a Trojan-injected neural network is shown in Figure 3.6. When the trigger condition is satisfied, the injected neurons will propagate the malicious behaviors to subsequent layers and finally modify the output prediction, as marked in red in Figure 3.6. Note that multiple Trojans need to be injected to achieve the attacking objective in most cases, as each operation in the network has a minor effect on the output within the dynamic range T, especially for deep neural networks.

Due to the layered structure, a neural network can be divided into sub-networks separated at the layer l, which can be expressed as

$$F(\mathbf{x}) = F_{l+1:L}(H_l);\ H_l = F_{l:l}(H_{l-1}) \text{ and } H_{l-1} = F_{0:l-1}(\mathbf{x}). \tag{3.1}$$

This modularity is further increased by the natural division of operations inside a network layer. For example, as shown in Figure 3.7, a dense layer is usually composed of multiplication operations followed by an accumulation operation and finally an activation function, plus any additional operations such as pooling. These operations create additional natural break points in which an adversary can inject Trojans.

To perform the hardware Trojan attack, the adversary also needs to pick an input trigger $\tilde{\mathbf{x}}$. In the proposed framework, the trigger can be chosen arbitrarily or similarly to legitimate inputs to achieve a higher degree of stealthiness. Then, we use the input trigger and the functions representing the first two sub-networks to obtain the intermediate values following the first and second

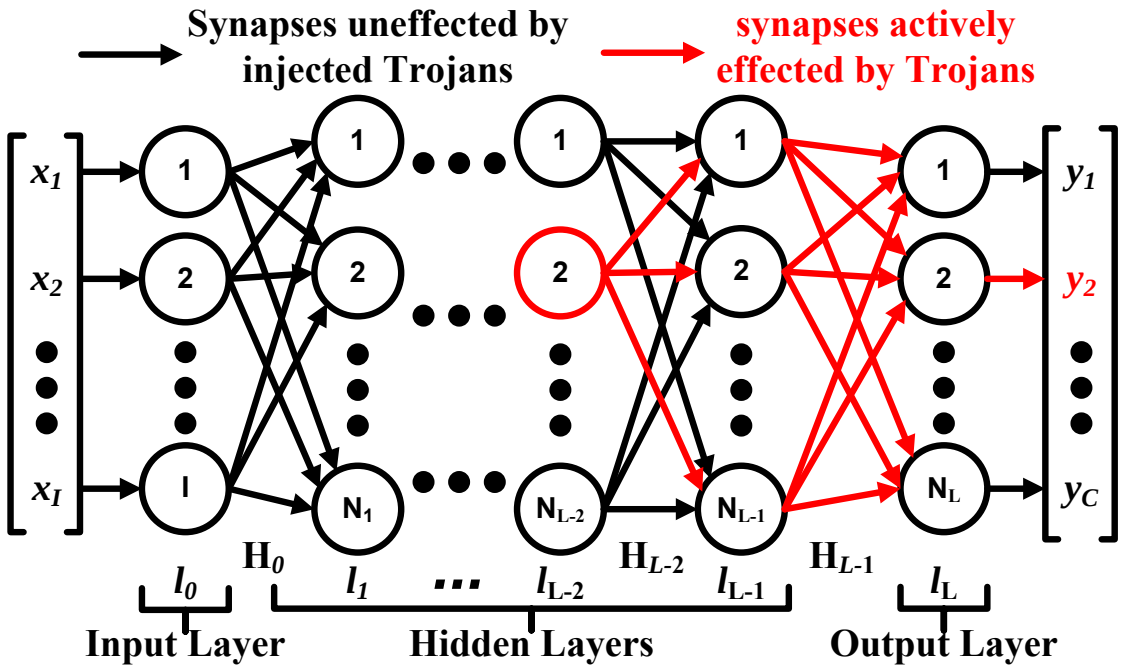

Figure 3.6: A neural network injected with hardware Trojans, the effect of the Trojans is propagating through some neurons but can be filtered out on others.

subnetworks, i.e., $H_{l-1} = F_{0:l-1}(\tilde{\mathbf{x}})$ and $H_l = F_{l:l}(H_{l-1})$. We then apply a modified adversarial sampling algorithm with respect to the target layer to find the perturbation needed to induce in the layer l to achieve either a targeted or untargeted attack. In order words, the goal is to generate $\tilde{H}_l = H_l + \mathbf{p}$ such that $F_{l+1:L}(\tilde{H}_l)$ is altered as intended, while the perturbation $\mathbf{p}$ for each modified neuron is bounded by the dynamic range based on the neural network model. Finally, the Trojan circuitry is designed according to the required perturbations and intermediate values.

3.2.2 Malicious Behavior Generation

While a variety of approaches in the existing literature for producing adversarial perturbations may be incorporated in the above framework, we choose to develop our approaches based on the JSMA algorithm [122], since it is designed specifically for minimizing the L_0-norm which could potentially minimize the total modified operations. The JSMA algorithm generates a Jacobian matrix with respect to the input and then utilizes the Jacobian with a set of rules to build a saliency map. By modifying the rules when constructing the saliency map, different adversarial objectives

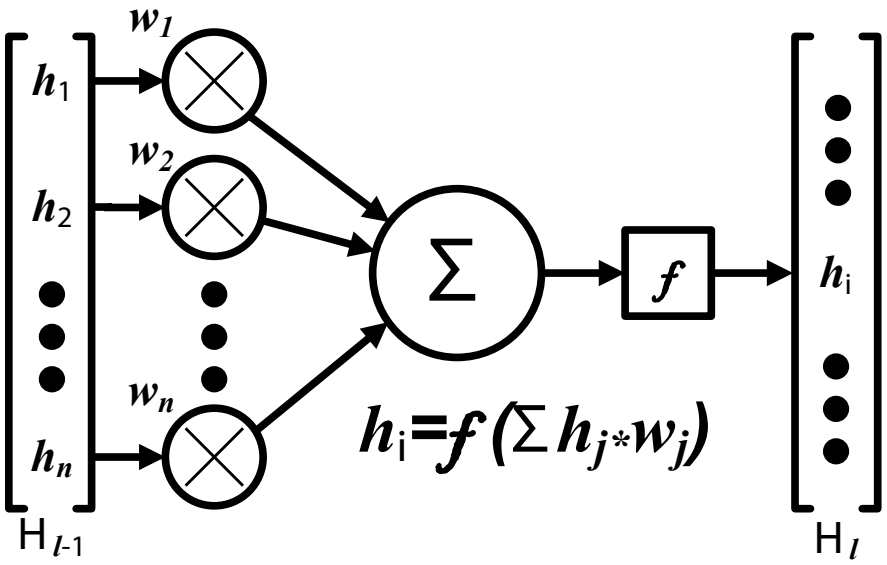

Figure 3.7: The basic hardware operations and function of a neuron.

can be prioritized.

As opposed to the original JSMA algorithm, in our proposed method, we modify the Jacobian as presented in Equation (3.2).

$$J(H_l) = \left[\frac{\delta F_{l+1:L}(H_l)[c]}{\delta H_l}\right]_{c \in C}. \tag{3.2}$$

Calculating the Jacobian begins at each output and is forward propagated to the target layer using the following modified version of the chain rule.

$$\frac{\delta H_{l_i}}{\delta H_l} = H_{l_i} \cdot W_{l_i} \cdot \frac{\delta H_{l_i - 1}}{\delta H_l} \tag{3.3}$$

Each column in the Jacobian corresponds to a specific output, while each row is linked to a specific neuron in the targeted layer. This is distinct from the original algorithm, as the rows of the original Jacobian matrix were linked to the input image. The element at the intersections of these rows and columns of this matrix indicate the strength of the correlation between the neuron/output pair. In this way, each entry of the Jacobian matrix indicates the correlation under the L_0-norm between the output classification and the intermediate value. It should be noted that each neuron is often tied to multiple outputs with varying strengths and so selecting the neurons should be done through a strict set of rules.

Consequently, a saliency map can be generated using the Jacobian matrix based on the specific goal of the adversary. An attacker with a targeted scope seeks to accomplish the goal in a

specific way, while the untargeted scope simply attempts to cause the specified input to misclassify to any other classes. In our methodology, we modify the rules for building the saliency map to incorporate both scopes. For instance, in an untargeted attack, the difference between the negative values in the column corresponding to the predicted class, $d\mathbf{x}_p^-$, and the positive values from every other column, $d\mathbf{x}_{i\neq p}^+$, can be used to form the saliency map:

$$S(\mathbf{x})[i] = \beta_p \left| d\mathbf{x}_p^- \right| + \beta_s \sum_{i \neq p} d\mathbf{x}_i^+. \tag{3.4}$$

Each entry in this map essentially indicates the effectiveness of simultaneously achieving the primary goal (i.e., decreasing the confidence of the predicted class) and the secondary goal (increasing the probability of a different class) by modifying the corresponding neuron. To gain optimal results in specific adversarial settings, β_p and β_s are introduced to weigh the primary and secondary goals.

Algorithm 1 Untargeted Hardware Trojan Attack

Require: $F(\cdot), \tilde{\mathbf{x}}, T, l$
1: $F(\cdot) \rightarrow F_{0:l-1}(\cdot)$, $F_{l:l}(\cdot)$ and $F_{l+1:L}(\cdot)$
2: $H_{l-1} = F_{0:l-1}(\tilde{\mathbf{x}})$
3: $\tilde{H}_l = H_l = F_{l:l}(H_{l-1})$
4: $\mathbf{y}_p = F_{l+1:L}(H_l)$
5: $L = [\ ]$
6: **while** $F_{l+1:L}(\tilde{H}_l) = \mathbf{y}_p$ and $|\mathbf{p}|_0 < T$ **do**
7: forward propagate $J(\tilde{H}_l)$
8: $S = Untargeted\_SM(J(\tilde{H}_l), \mathbf{y}_p)$, using Equation (3.4)
9: increase $\tilde{h}_n = argmax(S)$
10: $p = \tilde{H}_l - H_l$
11: **if** $\left|\tilde{h}_n\right|$ exceeds T **then**
12: $L.append(h_n)$
13: **end if**
14: **end while**
15: generate trigger design based on H_{l-1}
16: generate payload design using $\mathbf{p}$

However, the goal of a targeted attack is to decrease the probability of the targeted class over that of the predicted class. Consequently, decreasing the probability of the currently predicted class remains the primary goal, but the attack also incorporates an auxiliary goal of increasing the confidence of the target class. We also impose a secondary goal of keeping the remaining probabilities low. Thus, we formulate a saliency map using Equation (3.5).

$$S(\mathbf{x})[i] = \beta_p \left| d\mathbf{x}_p^- \right| + \beta_a d\mathbf{x}_t^+ + \beta_s \sum_{i \neq t,p} \left| d\mathbf{x}_i^- \right|, \tag{3.5}$$

Here we include three constants, β_p, β_a, and β_s, to weight the primary, auxiliary, and secondary

goals to gain optimal results in specific adversarial settings.

To find the modification needed in H_l, we perturb the operation associated with the largest values in the vector and modify them according to the adversarial objective. The magnitude of the perturbation should be bounded by the dynamic range T in the original neural network. For example, the value after the Trojan-injected neuron should be bounded between -1 and 1 if the activation is *tanh*. A *ReLU* activation function leads to a theoretically unbounded upper limit; however, in a practical real world attack any modifications would be limited due to the physical representation of the values. For the bounded attack, we use a bounding list, L, to lock neurons that cannot be further altered in the desired direction. We present the algorithms for the untargeted attack and the targeted attack in Algorithm 1 and Algorithm 2, respectively.

Algorithm 2 Targeted Hardware Trojan Attack

Require: $F(\cdot), \tilde{\mathbf{x}}, \tilde{\mathbf{y}}, T, l$
1: $F(\cdot) \rightarrow F_{0:l-1}(\cdot)$, $F_{l:l}(\cdot)$ and $F_{l+1:L}(\cdot)$
2: $H_{l-1} = F_{0:l-1}(\tilde{\mathbf{x}})$
3: $\tilde{H}_l = H_l = F_{l:l}(H_{l-1})$
4: $L = [\,]$
5: **while** $F_{l+1:L}(\tilde{H}_l) \neq \tilde{\mathbf{y}}$ and $|\mathbf{p}|_0 < T$ **do**
6: forward propagate $J(\tilde{H}_l)$
7: $S = Targeted\_SM(J(\tilde{H}_l), \tilde{\mathbf{y}})$, using Equation(3.5)
8: increase $\tilde{h}_n = argmax(S)$
9: $p = \tilde{H}_l - H_l$
10: **if** $\left|\tilde{h}_n\right|$ exceeds T **then**
11: $L.append(h_n)$
12: **end if**
13: **end while**
14: generate Trojan trigger design based on H_{l-1}
15: generate Trojan payload design using p

In addition to the original saliency map that indicates which neuron outputs to increase, we implemented the targeted attack with a second saliency map that indicates which neuron outputs to decrease. This slight modification allowed our implementation to mount attacks more quickly and efficiently than when utilizing only the single saliency map above.

3.2.3 Hardware Trojan Implementation

The implementation of the hardware Trojan design is highly dependent on the specific neural network architecture and the injected component of choice. Here, we only lay the groundwork and describe several possible designs. Note that other hardware Trojan designs of different types but

with similar functionalities can also be incorporated into the proposed framework.

The trigger of the hardware Trojan should be designed based on the internal state of the injected location, i.e., the produced H_{l-1} when feeding $\tilde{\mathbf{x}}$ through $F_{0:l-1}(\cdot)$. In addition, the triggerability must be extremely low to ensure the stealthiness of the hardware Trojan. A combinational circuit can be used to trigger the Trojan when even H_{l-1} closely resembles $\tilde{H}_{l-1}$. In the proposed framework, the payload should be designed to achieve the needed perturbation $\mathbf{p}(H_l)$ obtained from the malicious behavior generation step. We can either use a multiplexer logic which selects the output of malicious logic only when the Trojan is activated, or alter the internal structure of the certain operations to inject malicious behavior. For instance, several multipliers can be modified to produce rare outputs given the vector of H_{l-1}. We can also target on the activation function of each layer to directly alter H_l after the layer. Although our algorithms for malicious behavior generation are designed to minimize the hardware modification, we must still be careful in selecting the payload design such that the magnitude of change (e.g., the difference in side-channel leakage) is small enough to evade existing hardware Trojan detection techniques. The simplified block diagrams of two possible hardware Trojan designs are shown in Figure 3.8.

3.2.4 Injecting Hardware Trojans in a ReLU Function

Hardware Trojan modifications to basic arithmetic elements have been previously studied in the literature [129, 119]. In this section, as opposed to these common arithmetic elements, we present an efficient hardware Trojan design example using ReLU, which is a widely-used activation function nowadays. We consider a folded neural network design that is consistent with most of the practical implementations. Note that when the degree of parallelism is increased, less number of operations will need to be modified due to multiple operations being mapped to a single hardware block. The adversary may also choose to target other computational blocks in a neural network, such as a MAC or a pooling operation. The design methodology presented in this section can be easily extended to other blocks.

The function of a ReLU block can be expressed as

$$F(\mathbf{x}) = \begin{cases} \mathbf{x}, & \text{if } \mathbf{x} \geq 0 \\ 0, & \text{else} \end{cases} \tag{3.6}$$

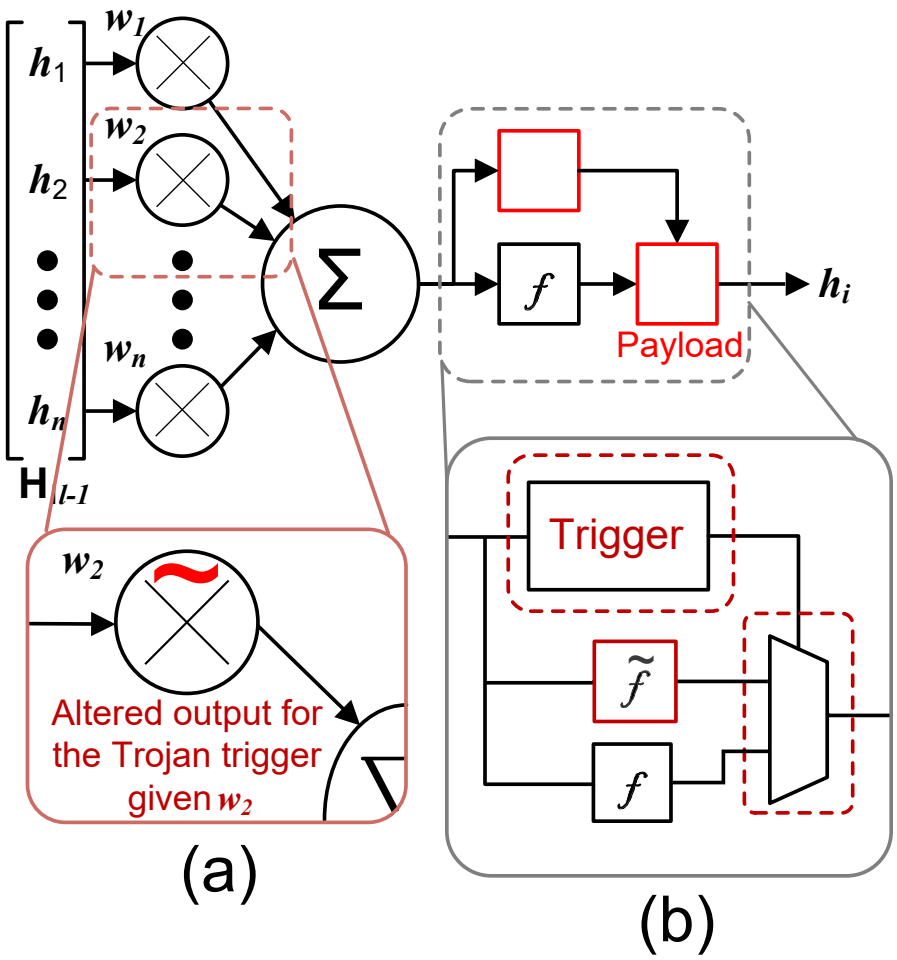

Figure 3.8: Simplified representations of two possible hardware Trojan designs on a neural network.

Its implementation involves combinational logic that decides whether to propagate the input or clear it to 0 according to the sign bit. An n-bit example with a simple Trojan is illustrated in Figure 3.9.

We design the detailed circuitry of the hardware Trojan based on the perturbation found in Section 3.2.2. Theoretically, the adversary can select any feasible input as the key for the Trojan activation. However, an input key will be more stealthy if it closely resembles the natural input distribution. Then, the state of the targeted operation can be determined by applying the input key to the network. We only select a small subset of the internal state, whose combination is rare across the operations that are time-multiplexed into the same block, as shown in Figure 3.9.

The payload is designed using simple logic gates to toggle the specific output bits. It is important to minimize the Trojan's impact when activated under a false trigger. For example, the

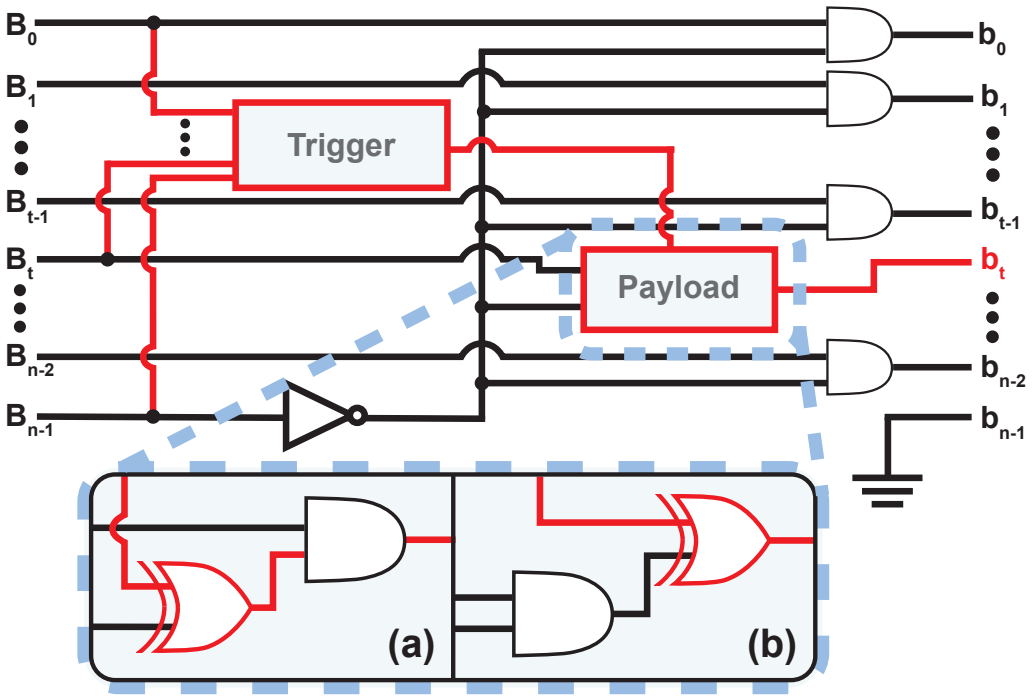

Figure 3.9: A ReLU implementation injected with a hardware Trojan, two possible payload designs are given.

single gate payload design, as shown in Figure 3.9(b), will alter the result of the ReLU block under every false trigger, while design in Figure 3.9(a) only alters the output when the corresponding input bit is 1. Additionally, the type of logic gate should be selected based on the values of the selected trigger and other false triggers. For example, we can generate another single gate payload design by replacing the XOR gate in the hardware Trojan design of Figure 3.9(b) with an OR gate. In this case, if a false trigger with an original output value of 1, the corresponding bit will not be altered even if the hardware Trojan is triggered. Thus, the impact of the false trigger can be effectively reduced. The other possible modifications to this single gate design are summarized in Table 3.1. While many of these single-gate payload designs are not stealthy alone, the payload can be expanded using additional logic gates to produce particular effects.

Trigger Output	Payload Functionality					
	AND	NAND	OR	NOR	XOR	XNOR
0	0	1	b_n	$\overline{b_n}$	b_n	$\overline{b_n}$
1	b_n	$\overline{b_n}$	1	0	$\overline{b_n}$	b_n

Table 3.1: Single Gate Payloads

We consider the operations that are folded into one single computational block as one folding set in a time-multiplexed architecture. According to the targeted perturbation generated from the modified adversarial example algorithm, the detailed hardware Trojans should be designed individually. The key idea is to isolate the targeted operation from other operations within the same folding set. We consider the following three scenarios.

1. One operation is modified in a folding set with one Trojan: In this scenario, the adversary can tailor the Trojan to the specific operation. According to both the targeted internal activation and required perturbation. The adversary can design the trigger with a pattern from several selected bits that are unique to the targeted operation.

2. Multiple operations are modified in a folding set with multiple Trojans: If perturbations are required to be applied to multiple neurons, it is likely that the triggering conditions or payloads are also different. In this scenario, the attacker can design the Trojans for each perturbation separately, which would yield better control of the stealthiness of the attack. However, a larger degree of hardware modification will be required.

3. Multiple operations are modified in a folding set with one Trojan: As opposed to the second scenario, the adversary can also aggregate all the perturbations into one hardware Trojan to reduce the magnitude of hardware modification. In this case, the trigger design is essentially a Boolean logic minimization problem that seeks to merge all the targeted internal activations. Unfortunately, this design results in the Trojan being less specific to each operation and thus increases the likelihood of false triggering and detection. However, the payload can be designed wisely to improve stealthiness. This can be done when the original values of the targeted operation and the false triggered operation are different, by using an appropriate combinational logic as described in Table 3.1. Additionally, the payload logic can be enhanced to alter the effect based on specific internal activations. For example, consider Figure 3.10, which alters the ReLU output based on the true output of the operation. While the two input bits may be required to be excluded in the trigger logic due to conflicting trigger conditions, these payload designs only allow bits to be altered based on that excluded condition.

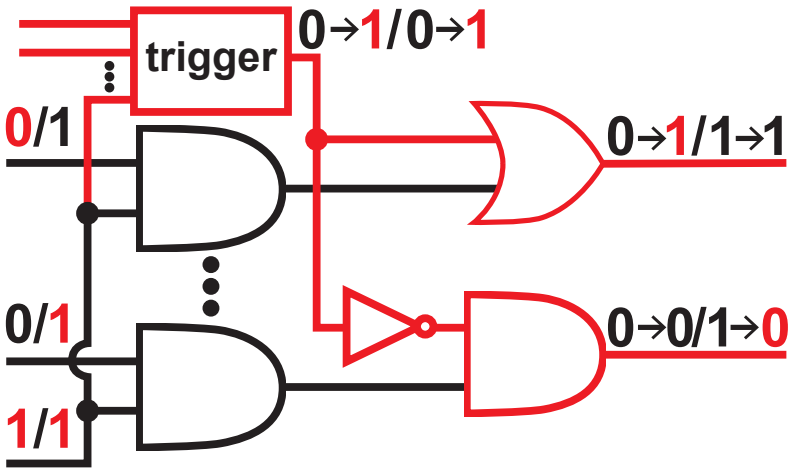

Figure 3.10: A payload designed to handle multiple perturbations. Two input scenarios are considered in the first the input combination "0...01" is applied to the input, and "1...11" in the second. Both sequences are detected by the trigger, which compared can be implemented by a simple comparator that evaluates specific bit patterns. This activates two payload circuits attacked to two outputs. The first payload only flips '0's to '1's, and so does not alter the second input. Likewise, the second payload only flips '1's to '0' and so does not alter the first input.

3.3 Experimental Evaluations

3.3.1 Software Simulations

3.3.1.1 Experimental Setup

We simulate the functionality of the Trojan by building a python instance of the network and passing the targeted input through it. Analyzing the binary representation of the inputs to the activation function of the target layer, we generate the logic required to activate the Trojan. Similarly, using the binary representation of the activation function's output, we determine the specific bits that need to be modified. The same Trojan logic is applied across all activation functions in the target layer. We then repeat this process over three target test cases.

In these experiments, we consider two different input trigger designs, i.e., **well-crafted** and **random** input triggers. Well-crafted input triggers are intended to achieve higher degrees of stealthiness against human observers by making the trigger very close to legitimate inputs. In order to ensure the similarity of the well-crafted input trigger, $\tilde{\mathbf{x}}$, to the test images, we randomly pull a single instance from the test set and form a new set for testing with the remaining samples. Randomized images adhering to the standards of the datasets are used as random input triggers. We

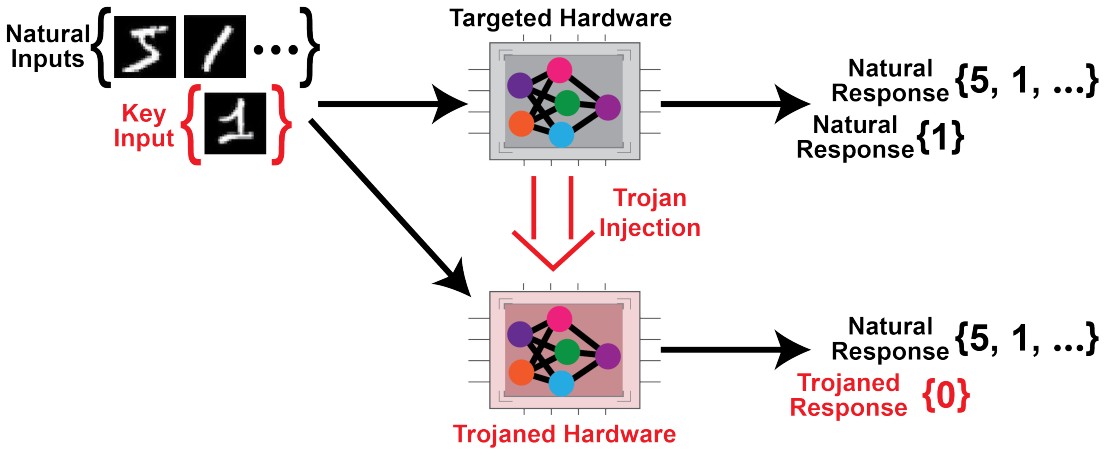

Figure 3.11: We experimentally inject a backdoor into the deep system through modifications in the hardware components. These modifications successfully able to alter the model's computation of a target key input with minimal hardware overhead while preserving the original functionality of the model in general.

evaluate both **targeted** and **untargeted** hardware Trojan attacks on the above neural networks. In our experiments, we use every other class of each dataset except the correct label as the targeted class for the targeted attack. While for the untargeted attack, we simply attempt to alter the prediction without a targeted class. The possible magnitude of perturbation generated by the payload is constrained by the dynamic range of the original benign model. We use $ReLU$ as the activation function on each layer when illustrating the **unbounded** scenario while using $tanh$ as the activation function for the **bounded** scenario. We examine the performance of hardware Trojan attacks on all hidden layers and the output layer. We show that the proposed method could inject hardware Trojans into any layer to generate malicious behavior and compare the effectiveness and stealthiness of different layers of choice.

3.3.1.2 Metrics

The success of a hardware Trojan attack is measured by its capabilities of altering the predictions and evading detection. In this work, we use effectiveness and stealthiness to evaluate these metrics. Additionally, we use the number of modified neurons to show the magnitude of change from the hardware implementation perspective. In sum, we define the following metrics:

- **effectiveness** (eff) is the percentage of input triggers classified as the intended label in either targeted or untargeted scenarios. An effective hardware Trojan attack should yield a

	MNIST				CIFAR-10			
	bound		unbound		bound		unbound	
layer	mfn (%)	eff (%)	mfn (%)	eff (%)	mfn (%)	eff (%)	mfn (%)	eff (%)
1	0.19	100	0.06	100	0.21	100	0.09	100
2	0.10	98	0.04	100	0.14	100	0.06	100
3	0.39	95	0.14	100	0.15	100	0.09	100
4	0.22	100	0.07	100	0.81	100	0.54	100
5	1.51	96	0.09	98	4.57	100	0.34	100
6	8.13	100	2.71	100	13.53	100	1.69	100
7	21.20	100	30.53	100	20.20	100	21.80	100

Table 3.2: Random input triggers for targeted attacks

high value (i.e., close to 100%) of this metric.

- **stealthiness (*stl*)** is the percentage of the outputs obtained from the Trojan-injected network that matches the predictions of the benign network. Ideally, this metric should be 100% to avoid detection by test data that are unknown to the adversary when returned to the designer.

- Number of **modified neurons (mfn)** is the average number of neurons that are modified to generate the desired malicious behavior. This metric directly correlates to the number of hardware modifications needed to implement the attack. This metric is also reported as the percentage of neurons modified in the targeted layer needed to achieve a misclassification.

3.3.1.3 Results and Discussion

The results of our experiments are summarized in Table 3.2 and Table 3.3. Note that each experiment is conducted with at least 1000 iterations, and the averages are presented. In addition, we test the stealthiness of the proposed methods using the test data of each dataset. **Our experimental results show that the proposed algorithms achieve** 100% **stealthiness for both datasets under all adversarial scenarios.** It can be seen from both Tables 3.2 and 3.3 that the percentage of modified neurons increases towards the latter layers of both networks. However, if we compare the absolute value of modified neurons, as shown in Figures 3.12 and 3.13, the latter layers actually require significantly fewer neurons to be modified to inject malicious behavior. Thus, injecting into neurons in the latter layers could result in a higher impact on the output, which conforms to our expectations. Note that the lowest possible percentage of modified neurons is 10%

for the output layer since it has a total of 10 neurons in both networks.

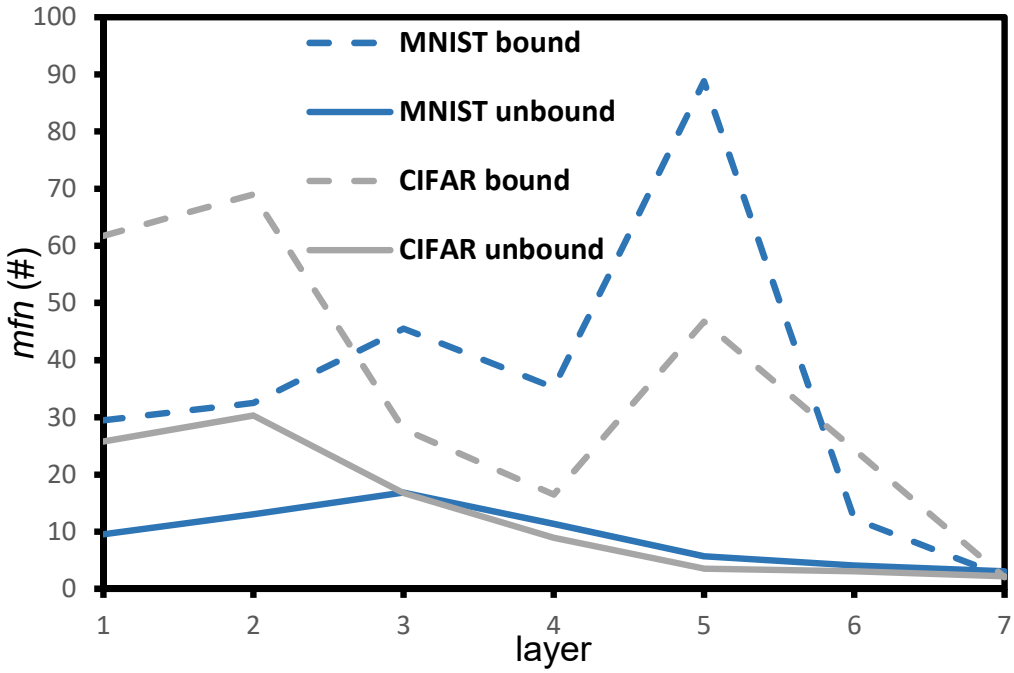

Figure 3.12: Number of modified neurons per layer given random trigger inputs in the targeted adversarial setting.

We first use random input triggers to mount the hardware Trojan attacks under the targeted adversarial scenarios. The results are presented in Table 3.2 and Fig. 3.12. It can be seen that the targeted attack under the unbounded scenario is stronger than the bounded scenario, as it requires fewer neurons to be modified. In other words, different neural network designs also lead to different levels of security from the hardware perspective. It also appears that both of these attacks perform well, reaching near 100% effectiveness on all layers while modifying only a small sample of the neurons. For example, our method could effectively classify a random input trigger as a specified class on the MNIST dataset by injecting hardware Trojans into 0.04%, on average, of neurons in the 2nd hidden layer of the neural network.

We next evaluate the performance of well-crafted input triggers on the datasets under the unbounded scenario. The results are illustrated in Table 3.3 and Fig. 3.13. It can be seen that these attacks also achieve very high effectiveness while modifying a small percentage of the neurons. For instance, our algorithm can effectively alter the classification of a well-crafted input image in an untargeted scenario while only altering, on average, 0.03% of the neurons in the 5th layer of an

	MNIST				CIFAR-10			
	targeted		untargeted		targeted		untargeted	
layer	mfn (%)	eff (%)	mfn (%)	eff (%)	mfn (%)	eff (%)	mfn (%)	eff (%)
1	0.18	100	0.17	100	0.34	97	0.28	100
2	0.06	98	0.08	100	0.53	100	0.43	100
3	0.14	98	0.04	100	0.53	100	0.36	100
4	0.09	100	0.13	100	0.66	100	0.45	100
5	0.39	99	0.03	100	0.90	100	0.15	100
6	1.83	100	0.67	100	2.11	97	0.55	100
7	22.17	100	10.00	100	29.30	97	10.00	100

Table 3.3: Well-crafted input triggers under the unbounded scenario

MNIST classifier. Under this scenario, it can be observed that the untargeted attack usually requires fewer modifications than the targeted attack since it has the flexibility to select the most accessible malicious output.

When comparing the results between the MNIST and CIFAR10 classifiers under the same adversarial settings, we can observe that attacks on the CIFAR10 classifier, in general, require a larger percentage of neurons to be modified. This is further compounded by the fact that the majority of the layers in the CIFAR10 classifier have more neurons than the corresponding MNIST classifier. For example, when targeting the 2nd layer, our algorithm only needs to modify less than 50 neurons of the MNIST classifier, while over 200 neurons have to be altered in the CIFAR10 classifier.

Finally, we observe that the adversary's choice of input trigger affects the strength of the attack. By comparing experimental results between the well-crafted and random input triggers of the CIFAR10 classifier, it is apparent that the attacks based on well-crafted input triggers require more modifications. Specifically, attacks on the second layer of the network require almost 9 times more modifications with well-crafted input triggers compared to random input triggers. Thus, a random input trigger could achieve higher stealthiness.

3.3.2 Hardware Synthesis

Finally, we ensure that an attacker can implement a Trojan in the hardware that exhibits the same functionality as the simulated models while limiting the size of the modification. To this end, we synthesize a Verilog implementation of a neuron in Sysnopsys' Design Compiler. The neuron

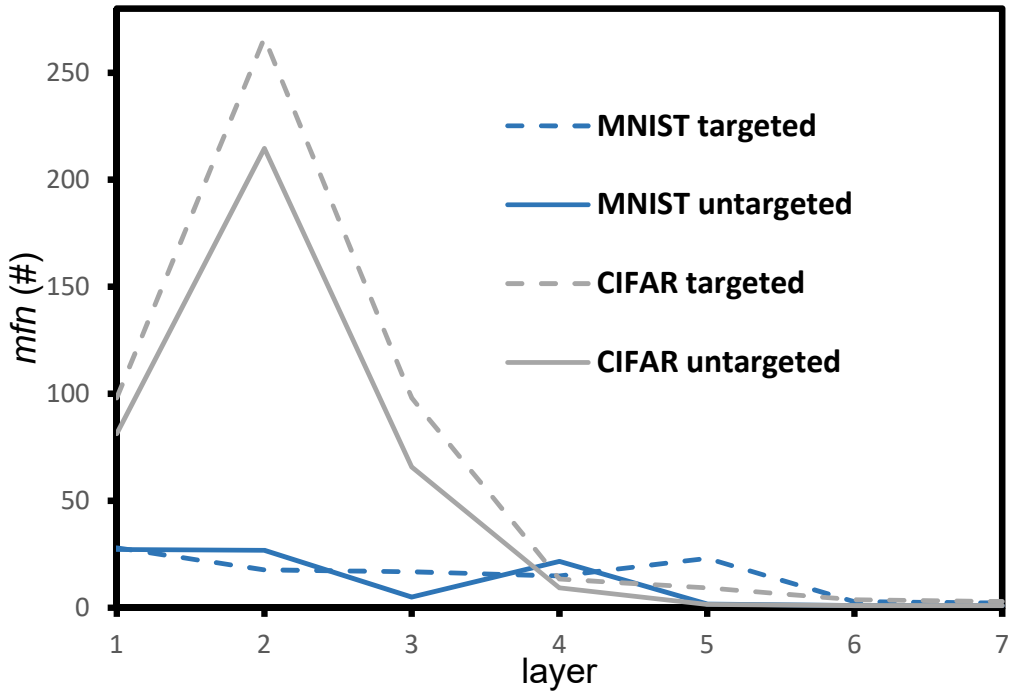

Figure 3.13: Number of modified neurons per layer given well-crafted trigger inputs in the unbounded adversarial setting.

contains the basic MAC and a ReLU function. We then alter the activation function's netlist to implement the same malicious modification seen in the Simulations.

3.3.2.1 Setup

To evaluate this Trojan injection technique, a similar experimental setup as in [29, 99] is employed. A seven-layer convolutional neural network is trained in TensorFlow to classify the MNIST dataset at an accuracy of over 99%. The operation targeted in the Trojan injection is the ReLU functions of a fully-connected layer. We use an input trigger in each iteration of our experiment that is arbitrarily selected from a single sample of the test data. We then randomly choose a targeted adversarial label for the input trigger and apply a modified L_0 norm-based adversarial example generation algorithm [122] to obtain the required perturbation.

We implement a time-multiplexed neural network architecture where the operations of each layer are folded into computational blocks. To reduce the size of the folding sets or increase the level of parallelism, architectures with multiple instances of the corresponding blocks are also examined.

The original architectures are synthesized using Synopsys Design Compiler that is mapped to a 32nm technology. In our experiment, we verify the effectiveness of the proposed hardware Trojan design under the worst-case scenario where the entire layer is composed of a single folding set. In other words, a Trojan would affect all the operations in that layer.

As described in Section 3.2.4, the detailed design of the hardware Trojan should be dependent on the internal activations of both the targeted operation and the false triggered operations in the folding set. For each case, we carefully select the input bits for the trigger design to reduce the rate of false triggering as well as appropriately design the payload to minimize the impact of false triggering. Then, we modify the netlist of the original architecture to simulate the hardware Trojan injection.

Scenario	Modified Operations	Injected Trojans	Detectable Images	Hardware Overhead
1	1	1	0%	0.0022%
2	2	2	0.05%	0.0136%
3	2	1	0.41%	0.0022%

Table 3.4: Summary of Experimental Results

3.3.2.2 Results

We determine the effectiveness of modification by assuring that the injected Trojan can produce the correct perturbations on the targeted operations. The effectiveness is verified by both the simulated network and the synthesized hardware architecture. Our experimental results show that the targeted misclassifications are 100% achieved by the injected hardware Trojan in all tested scenarios.

We then evaluate the stealthiness of the hardware Trojan design using test data of the MNIST dataset, where the input trigger is removed in each iteration. We compare the output classifications of the original design with the hardware Trojan-injected architecture. Our experimental results illustrate that the proposed hardware Trojan could achieve a very high degree of stealthiness. For example, the proposed Trojan designs are 100% undetectable on the test set in scenario 1, while scenarios 2 and 3 results in slightly lower stealthiness though the percentages of changed outputs are still within an acceptable range given the stochastic nature of neural network applications. Note that in practical application, the level of parallelism would be much higher. Thus, the stealthiness

of the hardware Trojan injection will also be much higher compared to our experimental setting.

Furthermore, we evaluate the feasibility of the Trojan injection by examining the hardware overhead needed to mount the attack. The results indicate that the hardware modifications in all scenarios are extremely tiny, which are nearly impossible to detect under the presence of manufacturing process variation. The experimental results are summarized in Table 3.4. As a result, we can conclude that the proposed hardware Trojan design is very effective and stealthy.

3.4 Conclusions

3.4.1 Summary

This work introduced, for the first time, hardware Trojan attacks in the scope of neural networks. In order to better define and understand hardware attacks, as well as other deep learning attacks that don't directly target the deep learning models, we expand the taxonomy of neural network security to include attacks conducted in the production phase.

In our experimental results, we verified the effectiveness of the proposed hardware Trojans through extensive software simulations and hardware synthesis. Our software simulations considered a variety of scenarios demonstrating targeted/untargeted attacks using both well-crafted and randomly input triggers. These experimental results demonstrated the stealthiness and effectiveness of the injected hardware Trojans. Additionally, the hardware modifications required to inject the malicious hardware were shown to be minimal.

One of the major limitations of this work is the requirement that the adversary knows the model that the developers will deploy to the hardware platform upon deployment. This requirement arises due to the need to connect the hardware modifications with the software running on the device, as without this connection, it is unclear how to produce a desired prediction outcome. Some follow-up works have arisen that attempt to overcome this limitation. One work explored the possibility of using the model as the Trojan trigger rather than the inputs [92]. Other works bypass this need by placing the Trojans in parallel with the hardware accelerator, making it model agnostic [167]. However, such solutions assume certain application scenarios and so can be applied in specific threat models.

Chapter 4

Preventing Deep Learning Hardware Piracy with Watermarks

This chapter presents DeepHardMark, the first watermarking framework for defending deep learning hardware from piracy. We presented this work at Association for the Advancement of Artificial Intelligence (AAAI) 2022.

4.1 Importance of Watermarking Deep Learning Hardware

While general-purpose processors are still widely utilized across the field [79], FPGA and ASIC solutions can provide superior performance and efficiency needed for critical commercial systems [111]. Nevertheless, modern horizontal supply chains often outsource fabrication, production, and distribution across multiple globalized corporations. Adversaries can take advantage of vulnerabilities in the supply chain to overproduce, copy, or recycle hardware designs for their own profit [90]. Therefore, it is critical to provide a means for hardware developers to assure the security of a design relinquished to the horizontal supply chain [166, 133]. Hardware watermarking allows designers to place a signature into their hardware intellectual properties (IPs) that verify rightful ownership [47, 127].

The outstanding accuracy of DNN systems comes at the cost of high computational complexity. As such, hardware accelerators for DNN inference have seen a resurgence in recent years [173,

128]. While GPUs and other high-performance computing platforms have enabled the widespread utilization of deep learning, the increasing demand for low-latency or low-power applications is driving a growing interest in more efficient platforms [145]. Premium DNN accelerators integrate high-volume computational arrays with well-orchestrated data flows that can maximize the utilization of hardware resources [144]. When a DNN is executed on the architecture, a mapper converts the algorithmic computations to hardware-compatible operations. Through careful consideration of the specific target scenario, IP developers generate efficient systems that can surpass general-purpose solutions [65].

Watermarking techniques are conventionally deployed as a countermeasure to multimedia IP theft [80]. Concern over the ease of DNN model theft has motivated researchers to extend these concepts to deep learning. To this end, researchers have leveraged model poisoning and backdoor attacks as a method of embedding the owner's signature into a model [176, 94]. This induces abnormal outputs for specific inputs that can identify the DNN. But such schemes are often circumventable by extending the defenses from the adversarial perspective [2, 172, 165]. DNN fingerprinting [70, 16] has also been investigated recently, which has a similar objective, i.e., IP ownership verification, but through extracting a fingerprint from a classifier without altering the model [16]. However, these prior works are not applicable to protecting private DNN hardware. Recent works have proposed hardware-assisted DNN obfuscation schemes to protect models [21, 23]. These methodologies are not targeted at identifying pirated models but at degrading performance when used fraudulently.

In this chapter, we present DeepHardMark, a watermarking methodology designed to defend deep learning accelerators from IP theft by introducing a backdoor into the hardware, which the rightful owner can reveal with a key DNN and key sample. The watermark signature is embedded solely through modifications to the hardware and so can be used as proof of ownership over a design. In addition, to the base algorithm, we also present optimizations on the algorithm, which can be used to improve the hardware modifications reducing the overhead and impact on the hardware's functionality.

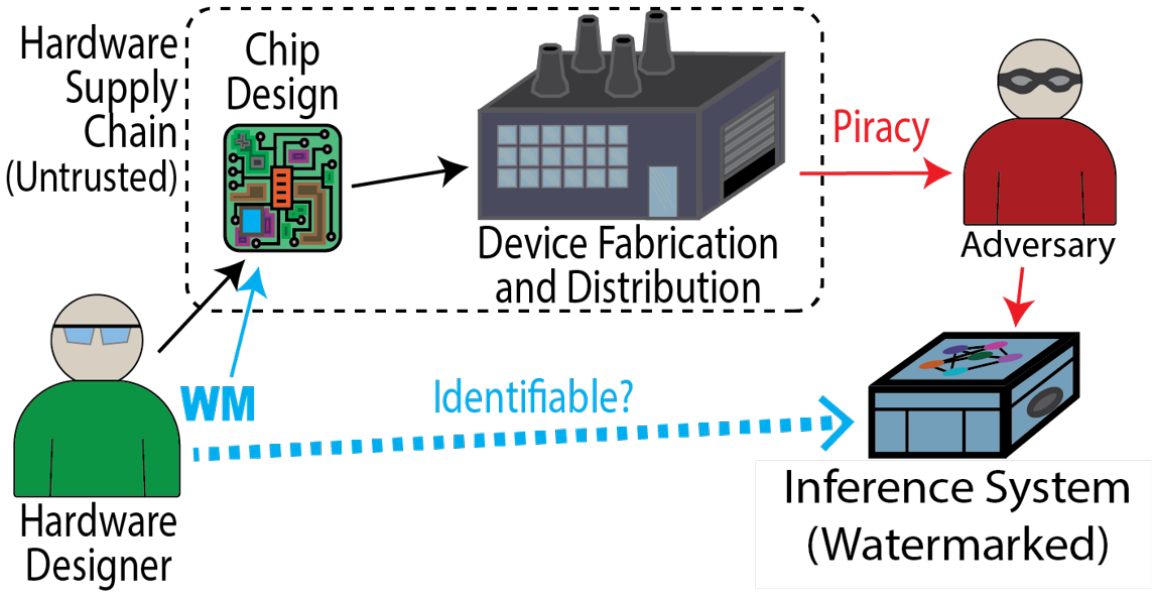

Figure 4.1: The high value of deep learning systems and vulnerability of the deep learning supply chain makes deep learning accelerators prime targets of piracy. A major tool for developers to defend these intellectual properties is through the use of a watermark. This defensive technique embeds a signature into the hardware design, which can be revealed during deployment to verify rightful ownership.

4.2 Embedding Watermarks in Deep Learning Hardware

4.2.1 Problem Setting

4.2.1.1 Threat Model

In this work, we consider a threat model that is consistent with the literature of hardware watermarking [135]. We assume that an adversary may attempt to pirate a DNN accelerator through the supply chain. For example, a malicious foundry may overproduce the devices and illegally sell them to other customers, or an adversary can attempt to make an illegal copy from a proprietary IP. As discussed above, building these IPs is non-trivial and involves a high cost, so adversaries have a strong economic incentive to steal an IP without paying the legitimate owner. Furthermore, previous schemes are targeted at verifying the algorithmic IPs and do not extend protection to the hardware. In alignment with prior works [32, 135], we assume the attacker does not have access to the behavioral description of the IP.

For the watermark verification, we consider a black-box setting, where after the deployment,

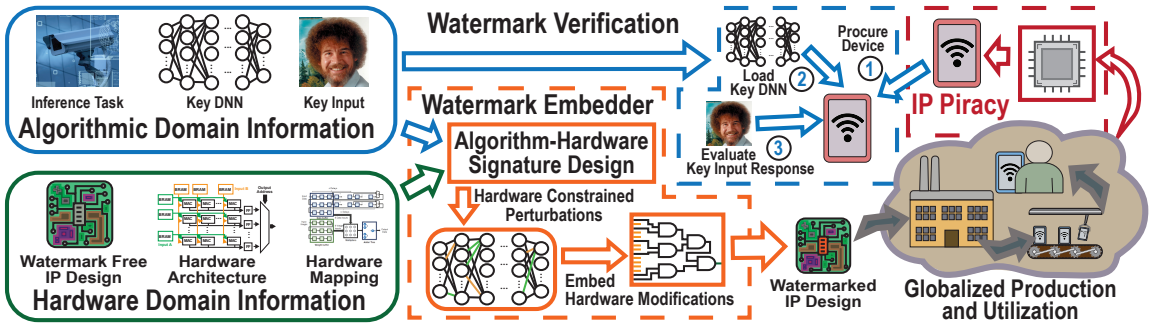

Figure 4.2: Overview of the proposed algorithm-hardware co-optimized watermarking methodology.

the IP owner will only be able to interact with the hardware through remote API calls, and any intermediate values are assumed to be unknown. The watermark should be embedded into the hardware such that its presence can be easily verified through the API. We also require that the system be general enough to accommodate and map different models for execution.

4.2.1.2 Problem Statement

This work proposes an algorithm-hardware co-optimized methodology for embedding a hardware watermark into DNN hardware accelerators, as illustrated in Figure 4.2. In order to watermark a hardware design, the IP owner needs to embed an identifiable signature into the design that can be verified after deployment. For algorithmic IPs, this has been done by embedding backdoors into a protected DNN, $F(\cdot)$, by altering the model's behavior on specific *key samples*, $\mathbf{x}_k$. Ideally, this *signature embedded model*, $F_P(\cdot)$, should only be altered for $\mathbf{x}_k$, described mathematically as:

$$F_P(\mathbf{x}) = \begin{cases} \mathbf{y}_k, & \text{when } \mathbf{x} = \mathbf{x}_k, \\ F(\mathbf{x}), & \text{otherwise,} \end{cases} \tag{4.1}$$

where it is required that $\mathbf{y}_k \neq F(\mathbf{x}_k)$. This can be done by altering the weights of $F(\cdot)$ to embed a signature in the DNN.

This work extends the DNN watermarking scheme into the hardware domain. This is accomplished by embedding modifications into the M functional blocks that execute the N operations in the DNN. These modifications alter the functionality of the DNN executed on the hardware without directly modifying the DNN itself. However, every modification to a specific functional block

will alter the computation of all operations executed on the block. As such, we introduce two binary matrices: the hardware mapping, $\mathbf{H} \in \{0,1\}^{M \times N}$, and a block selection mask, $\mathbf{B} \in \{0,1\}^{M}$, which identifies the hardware blocks targeted for modification. Using these structures, we compose the *block constrained perturbation*, $\boldsymbol{\delta}_k \in \mathbb{R}^{1 \times N}$, as:

$$\boldsymbol{\delta}_k = \boldsymbol{\delta} \odot \mathbf{BH}, \tag{4.2}$$

where $\odot$ signifies element-wise multiplication.

Equation (4.2) converts the unconstrained perturbation into a perturbation that describes the impact of hardware modifications on a DNN. In short, $\mathbf{B}$ factorizes $\boldsymbol{\delta}_k$ into groups of elements mapped to the different hardware blocks. By adjusting the elements of $\mathbf{B}$, we can enable or disable the perturbations caused by modifications to individual functional blocks. Then, by adjusting $\boldsymbol{\delta}$, we can determine the modifications needed in each functional block of the DNN. Our goal is to find a $\boldsymbol{\delta}_k$ that can alter the hardware's functionality on a *key DNN*, $F_k(\cdot)$, when evaluating the key samples, $\mathbf{x}_k$. We denote the execution of a model on hardware modified to generate a perturbation with a superscript. The hardware watermarking objective can be described by:

$$F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}) = \begin{cases} \mathbf{y}_k, & \text{when } \mathbf{x} = \mathbf{x}_k, \\ F_k(\mathbf{x}), & \text{otherwise,} \end{cases} \tag{4.3}$$

while any other DNNs executed on the hardware remains unchanged, i.e., $F^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}) = F(\mathbf{x})$. As embedding the modifications in the hardware does not require modifying the key DNN or key sample, the execution of $F_k(\mathbf{x}_k)$ on any unmodified hardware will produce the expected results from the algorithmic perspective. This is also a fundamental difference from prior DNN watermarking methods, which enables hardware verification.

As illustrated in Figure 4.2, to verify the design, the IP owner first accesses a stolen watermarked version of the hardware accelerators and the original watermark-free version. Then, the owner must load the key DNN, $F_k(\cdot)$, onto the hardware. First, establishing the functionality of both designs is demonstrably the same when executing $F_k(\cdot)$ over a dataset randomly drawn from the input domain. Then, the IP owner then compares the functionality of both designs when computing the key sample, $F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}_k) \neq F_k(\mathbf{x}_k)$. The owner can then identify the irregular behavior as an

embedded signature verifying ownership of the design. This verification procedure follows a scheme similar to those seen in the algorithmic perspective [62].

4.2.2 Proposed Methodology

4.2.2.1 High-level DeepHardMark Algorithm

The proposed method is mainly composed of three stages. First, we determine a *block constrained perturbation*, $\boldsymbol{\delta}_k$, that can produce the signature embedded model $F_k^{\boldsymbol{\delta}_k}(\cdot)$ by perturbing $F_k(\cdot)$. As the end goal is to embed these perturbations into the hardware, $\boldsymbol{\delta}_k$ is carefully crafted so that they are constrained to operations mapped to the same hardware blocks. To this end, as opposed to perturbing the weight of $F_k(\cdot)$, we introduce the perturbations on the functional blocks, as seen in previous hardware backdoor attacks [30]. We utilize a novel hardware-aware algorithm that constrains $\boldsymbol{\delta}_k$ based on the hardware mapping of the DNN's operations. We then minimize the effect of $\boldsymbol{\delta}_k$ within each hardware block by filtering out redundant perturbations to produce $\boldsymbol{\rho}_k$, the *operation reduced perturbation*. $\boldsymbol{\rho}_k$ defines which of the specific operations executed within the target hardware blocks which should be perturbed. Then, in the final stage of the algorithm, we can convert $\boldsymbol{\rho}_k$ into a *hardware modification set*, $\boldsymbol{\mu}_k$, that defines the specific trigger and payload signals. These modifications can then be embedded into the functional blocks to induce the desired behavior when executing $F_k(x_k)$.

4.2.2.2 Block Constrained Perturbations

The first step in the proposed methodology is to determine a set of perturbations, $\boldsymbol{\delta}_k$, seen in Equation 4.2. To minimize the number of hardware blocks that need to be modified, we craft $\boldsymbol{\delta}_k$ by targeting DNN operations executed on the same functional block. We can utilize the decomposition of $\boldsymbol{\delta}_k$, $\boldsymbol{\delta} \odot \mathbf{BH}$, as discussed in the previous section. A $\boldsymbol{\delta}_k$ that embeds the signature should satisfy the optimization problem:

$$\begin{aligned} \underset{\boldsymbol{\delta},\mathbf{B}}{\text{minimize}} \quad & L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k), \mathbf{y}_k), \\ \text{subject to} \quad & \mathbf{1}^T\mathbf{B} < \psi,\ \mathbf{B} \in \{0,1\}^M. \end{aligned} \tag{4.4}$$

Here L represents a loss function, such as cross-entropy loss, that quantifies the watermarking objective with respect to a target output, $\mathbf{y}_k$. $\mathbf{1}^T\mathbf{B} < \psi$ is a cardinality constraint that defines an upper bound on the number of hardware blocks that $\mathbf{B}$ selects to be perturbed. To ensure that

we find a minimal choice for $\mathbf{B}$, we are able to begin our search by using a large value for ψ and iteratively decrease it until a valid solution cannot be found. Because $\boldsymbol{\delta}$ is a continuous function, while $\mathbf{B}$ is a discrete integer, Equation (4.4) presents a Mixed Integer Programming (MIP) problem.

A methodology, known as ℓ_p-Box Alternating Direction Method of Multipliers (ℓ_p-ADMM), for solving such MIP problems has recently emerged [161]. This method has been broadly employed in many integer programming tasks for its superior performance [50, 177, 175]. Following this methodology, we decompose the integer constraint as: $\mathbf{B} \in \{0,1\}^M \Leftrightarrow \mathbf{B} \in \mathcal{S}_b \cap \mathcal{S}_p$ where $\mathcal{S}_b = [0,1]^M$ and $\mathcal{S}_p = \{\mathbf{B} : \|\mathbf{B} - \frac{1}{2}(\mathbf{1})\|_2^2 = \frac{M}{4}\}$. Detailed proof of this relationship can be found in the original paper [161]. Intuitively, these constraints define an ℓ_∞-box and corresponding ℓ_2-sphere which intersects the box only at its corners. These structures are carefully positioned so that their intersection contains only all binary combinations of $\mathbf{B}$. This substitution allows Equation (4.4) to be reformulated as a continuous representation of the MIP problem:

$$\begin{aligned} &\underset{\boldsymbol{\delta},\mathbf{B},\mathbf{S}_1\in\mathcal{S}_p,\mathbf{S}_2\in\mathcal{S}_b}{\text{minimize}} \; L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x_k}),\mathbf{y_k}), \\ &\text{subject to} \quad \mathbf{1}^T\mathbf{B} < \psi,\ \mathbf{B} = \mathbf{S}_1,\ \mathbf{B} = \mathbf{S}_2, \end{aligned} \tag{4.5}$$

where $\mathbf{S}_1 \in \mathcal{S}_p$ and $\mathbf{S}_2 \in \mathcal{S}_b$. Because of the element-wise product between $\boldsymbol{\delta}$ and $\mathbf{BH}$, this problem can iteratively solved by alternating between fixing one variable while optimizing the other, as seen in Algorithm 3.

First, we initialize $\mathbf{B}$ to $\mathbf{1}$ and fix its value. This allows Equation (4.5) to be simplified to:

$$\underset{\boldsymbol{\delta}}{\text{minimize}} \quad L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k),\mathbf{y}_k). \tag{4.6}$$

This is a standard optimization problem similar to those seen across the field of machine learning, which can be solved using simple gradient descent-based methods by iteratively updating $\boldsymbol{\delta}$ according to Equation (4.7):

$$\boldsymbol{\delta} = \boldsymbol{\delta} - \epsilon_\delta \left[\frac{\partial L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k),\mathbf{y}_k)}{\partial \boldsymbol{\delta}} \right]. \tag{4.7}$$

Here ϵ_δ is a learning rate used to control the speed of convergence during gradient descent.

Algorithm 3 Block Constrained Perturbations

Require: $F_k(\cdot)$, $L(\cdot)$, $\mathbf{H}$, x_k, y_k
Hyperparameters: ψ, T_δ, T_B, ϵ_δ, ϵ_B, ρ_1, ρ_2, ρ_3
Ensure: $F_k(\mathbf{x}_k) \neq \mathbf{y}_k$

$\mathbf{B} = \mathbf{1}$; $\boldsymbol{\delta} = \mathbf{0}$
while $\mathbf{1}^T\mathbf{B} > c$ or $F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k) \neq \mathbf{y}_k$ **do**
  for $i \in [1, T_\delta]$ **do**
    $\boldsymbol{\delta} = \boldsymbol{\delta} - \epsilon_\delta \left[\frac{\partial L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k)}{\partial \boldsymbol{\delta}}\right]$
  end for
  $\mathbf{Z}_1 = \mathbf{Z}_2 = \mathbf{1}$; $\mathbf{Z}_3 = \mathbf{1}$
  for $i \in [1, T_B]$ **do**
    $\mathbf{S}_1 = \mathcal{P}_{\mathcal{S}_p}(\mathbf{B} + \frac{1}{\rho_1}\mathbf{Z}_1)$
    $\mathbf{S}_2 = \mathcal{P}_{\mathcal{S}_b}(\mathbf{B} + \frac{1}{\rho_2}\mathbf{Z}_2)$
    $\mathbf{B} = \mathbf{B} - \epsilon_B \left[\frac{\delta\mathcal{L}}{\delta\mathbf{B}}\right]$ {$\mathcal{L}$ is defined in Equation (4.9)}
    Update the dual parameters using Equation (4.16)
  end for
end while
$\boldsymbol{\delta}_k = \boldsymbol{\delta} \odot \mathbf{BH}$
return $\boldsymbol{\delta}_k$

Second, for a fixed value of $\boldsymbol{\delta}$, Equation (4.5) simplifies to

$$\begin{aligned} &\underset{\mathbf{B},\mathbf{S}_1\in\mathcal{S}_p,\mathbf{S}_2\in\mathcal{S}_b}{\text{minimize}} \quad L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k), \mathbf{y}_k), \\ &\text{subject to} \quad \mathbf{1}^T\mathbf{B} < \psi,\ \mathbf{B} = \mathbf{S}_1,\ \mathbf{B} = \mathbf{S}_2. \end{aligned} \tag{4.8}$$

This optimization problem should be solved by using the ADMM. The augmented Lagrangian function of Equation (4.8) can be expressed as:

$$\begin{aligned} &\mathcal{L}(\mathbf{B}, \mathbf{S}_1, \mathbf{S}_2, \mathbf{Z}_1, \mathbf{Z}_2, \mathbf{Z}_3) = L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k), \mathbf{y}_k) \\ &+ (\mathbf{Z}_1)^T(\mathbf{B} - \mathbf{S}_1) + (\mathbf{Z}_2)^T(\mathbf{B} - \mathbf{S}_2) + \frac{\rho_1}{2}\|\mathbf{B} - \mathbf{S}_1\|_2^2 \\ &+ \frac{\rho_2}{2}\|\mathbf{B} - \mathbf{S}_2\|_2^2 + \frac{\rho_3}{2}(\mathbf{1}^T\mathbf{B} - \psi) + h_1(\mathbf{S}_1) + h_2(\mathbf{S}_2). \end{aligned} \tag{4.9}$$

Here $\mathbf{Z}_1 \in \mathbb{R}^M$, $\mathbf{Z}_2 \in \mathbb{R}^M$, and $\mathbf{Z}_3 \in \mathbb{R}^1$ are dual variables with corresponding penalty parameters: ρ_1, ρ_2, and ρ_3. While $h_1(\mathbf{S}_1)$ and $h_2(\mathbf{S}_2)$ are boolean valued functions that return 1 when $\mathbf{S}_1 \in \mathcal{S}_p$ or $\mathbf{S}_2 \in \mathcal{S}_b$, and 0 otherwise.

The first step in solving Equation (4.8) is to update $\mathbf{S}_1$ by solving:

$$\mathbf{S}_1 = \underset{\mathbf{S}_1\in\mathcal{S}_p}{\text{argmin}}\ (\mathbf{Z}_1)^T(\mathbf{B} - \mathbf{S}_1) + \frac{\rho_1}{2}\|\mathbf{B} - \mathbf{S}_1\|_2^2. \tag{4.10}$$

Projecting the unconstrained solution into $\mathcal{S}_p$, we get:

$$\mathbf{S}_1 = \mathcal{P}_{\mathcal{S}_p}(\mathbf{B} + \frac{1}{\rho_1}\mathbf{Z}_1). \tag{4.11}$$

A standard solution when projecting to the ℓ_∞-box is to clip all values back within the space using $\mathcal{P}_{\mathcal{S}_p}(\mathbf{S}) = max(min(\mathbf{S}, \mathbf{1}), \mathbf{0})$.

Second, $\mathbf{S}_2$ is updated by minimizing Equation (4.12):

$$\mathbf{S}_2 = \underset{\mathbf{S}_2 \in \mathcal{S}_b}{\operatorname{argmin}} \ (\mathbf{Z}_2)^T(\mathbf{B} - \mathbf{S}_2) + \frac{\rho_2}{2}\|\mathbf{B} - \mathbf{S}_2\|_2^2. \tag{4.12}$$

Similar to $\mathbf{S}_1$, this can be found by projecting the unconstrained solution back onto $\mathcal{S}_b$.

$$\mathbf{S}_2 = \mathcal{P}_{\mathcal{S}_b}(\mathbf{B} + \frac{1}{\rho_2}\mathbf{Z}_2). \tag{4.13}$$

where $\mathcal{P}_{\mathcal{S}_b}(\mathbf{S}) = \frac{\sqrt{M}}{2}\frac{\mathbf{S}-0.5(\mathbf{1})}{\|\mathbf{S}-0.5(\mathbf{1})\|} + \frac{1}{2}(\mathbf{1})$.

Next, $\mathbf{B}$ is updated by perturbing the variable according to the augmented Lagrangian function, $\mathcal{L}$, as below.

$$\mathbf{B} = \mathbf{B} - \epsilon_B \left[\frac{\delta \mathcal{L}}{\delta \mathbf{B}}\right], \tag{4.14}$$

where

$$\begin{aligned}\frac{\delta \mathcal{L}}{\delta \mathbf{B}} = \frac{\delta L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x}_k), \mathbf{y}_k)}{\delta \mathbf{B}} + \rho_1(\mathbf{B} - \mathbf{S}_1) + \mathbf{Z}_1 \\ + \rho_2(\mathbf{B} - \mathbf{S}_2) + \mathbf{Z}_2 + (\rho_3(\mathbf{1}^T\mathbf{B} - \psi) + \mathbf{Z}_3)\mathbf{1}.\end{aligned} \tag{4.15}$$

Finally, we update the dual variables with:

$$\begin{aligned}\mathbf{Z}_1 &= \mathbf{Z}_1 + \rho_1(\mathbf{B} - \mathbf{S}_1) \\ \mathbf{Z}_2 &= \mathbf{Z}_2 + \rho_2(\mathbf{B} - \mathbf{S}_2) \\ \mathbf{Z}_3 &= \mathbf{Z}_3 + \rho_3(\mathbf{1}^T\mathbf{B} - \psi),\end{aligned} \tag{4.16}$$

before recomputing $\mathbf{S}_1$ and $\mathbf{S}_2$ and perturbing $\mathbf{B}$ until a valid solution for Equation (4.8) is found. We iteratively improve $\boldsymbol{\delta}_k$ by alternating between optimizing Equation (4.6) and Equation (4.8) as seen in Algorithm 3.

Algorithm 4 Reducing Intra-block Perturbations

Require: $\boldsymbol{\delta}_k$, $L(\cdot)$, $F(\cdot)$, C
1: $\mathbf{R}_\rho = \{\mathbf{0}\}$
2: $\mathbf{R}_N = \{\mathbf{R}_n |\ \|\mathbf{R}_n\|_\infty = 1,\ \mathbf{1}^T\mathbf{R}_n = 1,\ \mathbf{R}_n \odot \boldsymbol{\delta}_k \neq 0\}$
3: **while** $F_k^{\mathbf{R}_r \odot \boldsymbol{\delta}_k}(\mathbf{x}_k) \neq \mathbf{y}_k\ \forall\ \mathbf{R}_r \in \mathbf{R}_\rho$ **do**
4: $\quad \mathbf{R}_{\rho_N} = \{\mathbf{R}_r + \mathbf{R}_n | \mathbf{R}_r \odot \mathbf{R}_n = \mathbf{0}, \mathbf{R}_r \in \mathbf{R}_\rho, \mathbf{R}_n \in \mathbf{R}_N\}$
5: $\quad$ **Loss** $= [\ ]$
6: $\quad$ **for** $\mathbf{R}_{rn} \in \mathbf{R}_{\rho_N}$ **do**
7: $\quad\quad l_{rn} = L(F_k^{\mathbf{R}_{rn} \odot \boldsymbol{\delta}_k}(\mathbf{x}_k), \mathbf{y}_k)$
8: $\quad\quad$ **Loss**.append($(\mathbf{R}_{rn}, l_{rn})$)
9: $\quad$ **end for**
10: $\quad$ sort_by_loss(**Loss**)
11: $\quad \mathbf{R}_\rho = \{$**Loss**$[0 : C-1][0]\}$
12: **end while**
13: $\mathbf{R} = \underset{\mathbf{R}_r \in \mathbf{R}_\rho}{\operatorname{argmin}}\ L(F_k^{\mathbf{R}_r \odot \boldsymbol{\delta}_k}(\mathbf{x}_k), \mathbf{y}_k)$
14: **return** $\mathbf{R}$

4.2.2.3 Intra-block Perturbation Reduction: Search Based Approach

The *block constrained perturbation*, $\boldsymbol{\delta}_k$, is targeted at minimizing the number of hardware blocks perturbed by the watermarking algorithm. However, it does not constrain the total perturbation within these groupings. Thus, it is likely that redundant perturbations that contribute little to the watermark's performance are contained in $\boldsymbol{\delta}_k$. Thus, the next step in the algorithm removes these redundant perturbations finding a minimal subset of the perturbations from $\boldsymbol{\delta}_k$ required to embed the watermark.

We can mathematically define $\boldsymbol{\rho}_k = \mathbf{R} \odot \boldsymbol{\delta}_k$, an *operation reduced perturbation*, where $\mathbf{R} \in \{0,1\}^N$ specifies which perturbations to keep. We solve for $\mathbf{R}$ using:

$$\begin{aligned} \underset{\mathbf{R}}{\text{minimize}} \quad & \|\mathbf{1}^T\mathbf{R}\|, \\ \text{subject to} \quad & F_k^{\mathbf{R} \odot \boldsymbol{\delta}_k}(\mathbf{x}_k) = \mathbf{y}_k. \end{aligned} \tag{4.17}$$

We solve this problem by iteratively selecting the elements of $\boldsymbol{\delta}_k$ with the greatest impact on the objective function and including them in the $\boldsymbol{\rho}_k$ by enabling them with $\mathbf{R}$. The algorithm used to search for the $\boldsymbol{\rho}_k$ is inspired by the beam search algorithms commonly seen in natural language processing [109].

The search algorithm begins with two sets: $\mathbf{R}^\rho = \mathbf{0}$ and $\mathbf{R}_N = \{\mathbf{R}_n |\ \|\mathbf{R}_n\|_\infty = 1,\ \mathbf{1}^T\mathbf{R}_n = 1,\ \mathbf{R}_n \odot \boldsymbol{\delta}_k \neq 0\}$. We can understand $\mathbf{R}_N$ as the set of all meaningful single-bit iterations of $\mathbf{R}$. The algorithm's goal is to iteratively incorporate members from $\mathbf{R}_N$ into $\mathbf{R}_\rho$ by selecting the most

efficient choice at each step of the algorithm. We do this by generating the cartesian sum of both sets and determining which the choice of $\mathbf{R}^r \in \mathbf{R}^\rho$ and $\mathbf{R}^n \in \mathbf{R}^N$ best minimizes the loss function, $L(F_k^{(\mathbf{R}^r+\mathbf{R}^n)\odot\boldsymbol{\delta}_k}(\mathbf{x}_k), \mathbf{y}_k)$. These choices are then used to populate $\mathbf{R}^\rho$ during the next iteration of the algorithm, iteratively increasing the number of bits selected by the members of $\mathbf{R}^\rho$. Further, so that we don't sacrifice finding a superior solution by selecting the best choice at each iteration, we incorporate beam search techniques by keeping the top C choices for $\mathbf{R}^\rho$ rather than only the best. Algorithm 4 presents our implementation of this process.

4.2.2.4 Intra-block Perturbation Reduction: Gradient Decent Based Approach

The search-based approach to finding $\boldsymbol{\delta}_k = \mathbf{R} \odot \widehat{\boldsymbol{\delta}}_k$ is a computationally expensive process compounded by the need to maintain a list of the C-best solutions to decrease the likelihood that the algorithm converges sub-optimally. Using this algorithm limits DeepHardMark's usefulness when defending larger-scale models with many parameters. For the DeepHardMark$^+$ algorithm, we utilize an alternative gradient-based methodology for finding $\widehat{\boldsymbol{\delta}}_k$.

We begin by recognizing that we know a solution to $F_k^{\mathbf{R}\odot\widehat{\boldsymbol{\delta}}_k}(\mathbf{x}_k) = \mathbf{y}_k$ exists when $\mathbf{R} \odot \widehat{\boldsymbol{\delta}}_k = \widehat{\boldsymbol{\delta}}_k$, i.e. $\mathbf{R} = \mathbf{1}$. However, this stage of the algorithm aims to minimize $\|\mathbf{1}^T\mathbf{R}\|$, the number of operations targeted by the hardware modifications. Using a loss function, L, we can redefine the objective from Equation (4.17) in the form:

$$\begin{aligned} \underset{\mathbf{R}}{\text{minimize}} \quad & L(F_k^{\mathbf{R}\odot\widehat{\boldsymbol{\delta}}_k}(\mathbf{x}_k), \mathbf{y}_k) \\ \text{subject to} \quad & \mathbf{1}^T\mathbf{R} < \varepsilon, \end{aligned} \tag{4.18}$$

$$\mathbf{R} \text{ - Perturbation Reduction Mask} \tag{4.19}$$

where ε is an upper bound on the size of $\mathbf{R}$. Finding the optimal $\mathbf{R}$ implies finding the tightest bound on ε with a valid solution to (4.18). Solving for Equation (4.18) is analogous to finding the sparse solution to the objective function: $L(F_k^{\mathbf{R}\odot\widehat{\boldsymbol{\delta}}_k}(\mathbf{x}_k), \mathbf{y}_k)$.

To solve this, we perform an iterative algorithm using the gradient information. The proposed methodology can be seen in Algorithm 5. We begin with the initialization $\mathbf{R} = \mathbf{0}$. We then iteratively activate bits in $\mathbf{R}$ by first calculating $L(F_k^{\mathbf{R}\odot\widehat{\boldsymbol{\delta}}_k}(\mathbf{x}_k), \mathbf{y}_k)$ using $\mathbf{R}$. $\mathbf{R}$ is used as a mask to turn off individual elements of the unreduced perturbation, $\widehat{\boldsymbol{\delta}}_k$. The gradient of L with respect

Algorithm 5 Gradient-Based Intra-Block Reduction

Require: $\widehat{\delta}_k$, $L(\cdot)$, $F(\cdot)$, x_k, y_k
1: $\mathbf{R} = \mathbf{0}$
2: **while** $M \neq m$ **do**
3: **while** $F_k^{\mathbf{R}\odot\widehat{\delta}_k}(x_k) \neq y_k$ **do**
4: $l = L(F_k^{\mathbf{R}\odot\widehat{\delta}_k}(x_k), y_k)$
5: $M = \text{argmax}(\frac{\delta l}{\delta \mathbf{R}} \odot \mathbf{BH})$
6: $\mathbf{R}[M] = 1$
7: **end while**
8: **while** $F_k^{\mathbf{R}\odot\widehat{\delta}_k}(x_k) = y_k$ **do**
9: $l = L(F_k^{\mathbf{R}\odot\widehat{\delta}_k}(x_k), y_k)$
10: $m = \text{argmin}(\frac{\delta l}{\delta \mathbf{R}} \odot \mathbf{BH})$
11: $\mathbf{R}[m] = 0$
12: **end while**
13: **end while**
14: $\mathbf{R}[M] = 1$
15: **return** $\mathbf{R}$

to $\mathbf{R}$ is a valid local approximation of the impact switching bits in the mask on the loss. As such, we find the element with maximal effect on the loss with $M = \text{argmax}(\frac{\delta L}{\delta \mathbf{R}} \odot \mathbf{BH})$. We can then iteratively activate those bits in $\mathbf{R}$ until we arrive at a valid solution for Equation (4.18). Once a $\mathbf{R}$ is selected, which produces the desired mapping $F_k^{\mathbf{R}\odot\widehat{\boldsymbol{\delta}}_k}(\mathbf{x}_k) = \mathbf{y}_k$ this mask can be used to extract a subset of $\widehat{\boldsymbol{\delta}}_k$ which produces the watermark signature.

However, as highlighted by the sparse adversarial example literature [42], such methods often introduce unnecessary or redundant elements to $\boldsymbol{\delta}_k$. To ensure the selection of the best mask, we perform additional rounds of optimization by activating features with minimal impact on the loss function, identified by $m = \text{argmin}(\frac{\delta l}{\delta \mathbf{R}} \odot \mathbf{BH})$. Then, we deactivate elements until the watermark signature is removed, i.e., $F_k^{\mathbf{R}\odot\widehat{\boldsymbol{\delta}}_k}(\mathbf{x}_k) \neq \mathbf{y}_k$. We repeatedly activate the necessary and deactivate unnecessary elements of $\mathbf{R}$ until both branches of the algorithm toggle the same set of features. This process ensures that we find a $\mathbf{R}$ which satisfies Equation (4.18) for a minimal choice of ε.

4.2.2.5 Hardware Watermark Modifications

It has been demonstrated that the hardware Trojans can be successfully leveraged to embed watermarks into a hardware design for conventional circuits [135]. Inspired by this, we convert the operation reduced perturbation, $\boldsymbol{\rho}_k$, to a *hardware modification set*, $\boldsymbol{\mu}_k$. Rather than a static perturbation applied to all inputs, it identifies the perturbation as a target trigger signal for activating the watermark and a target signal for the payload functionality that should be induced in operation.

A trigger and payload can then be designed around this information and embedded in the target functional block to produce the watermarked hardware $H_{\boldsymbol{\mu}_k}(\cdot)$. The specific design depends on the target hardware block and the stealth objectives of the designer. As a case study, our implementation embeds small combinational logic circuits into the target hardware, as shown in Figure 4.3. In our example, $\boldsymbol{\mu}_k$ contains observed binary input patterns to an operation when computing $\mathbf{x}_k$, and bit flip patterns that can produce the perturbation.

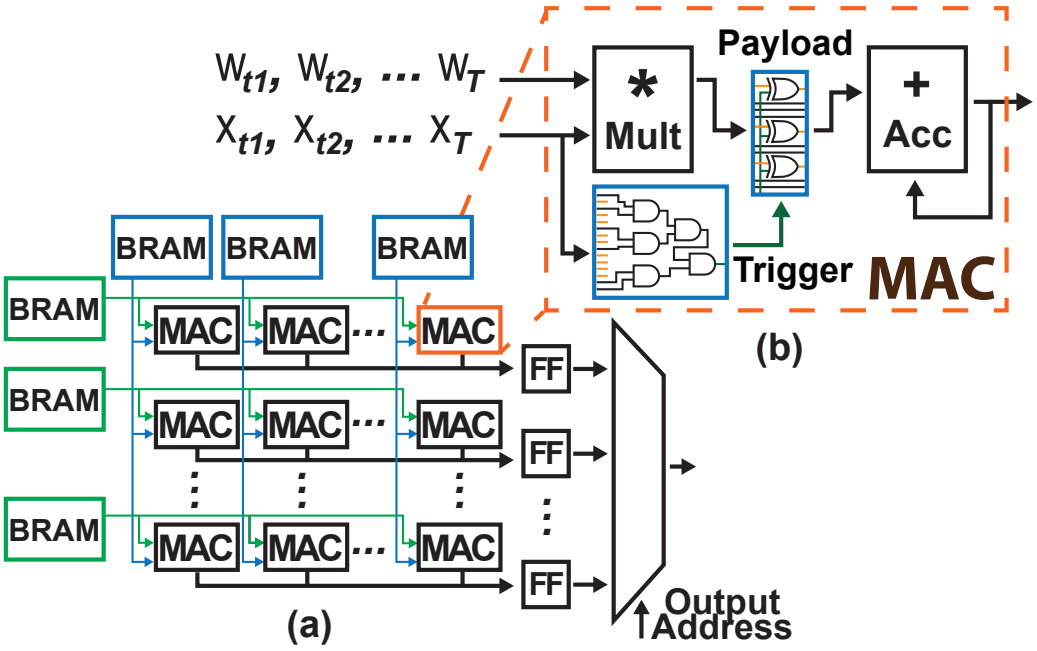

Figure 4.3: (a) A convolutional neural network hardware accelerator derived from [171]. (b) We can embed small combinational circuits into the hardware blocks of the IP. These circuits detect the target input combinations and flip the corresponding output bits as specified by $\boldsymbol{\mu}_k$.

To generate these modifications, we convert $\boldsymbol{\delta}_k$ into a *hardware modification set*, $\boldsymbol{\mu}_k$, which defines $\boldsymbol{\delta}_k$ in terms of modifications that can be embedded in the hardware. As a case study, our implementation embeds small combinational circuits into the target hardware, as shown in Figure 4.3. Let $op \in Ops$ be a target operation indicated by $\boldsymbol{\delta_k}$ with $O_k = |\boldsymbol{\delta}_k|$. In our example, $\boldsymbol{\mu}_k = \{\mathbf{P}_k, \mathbf{F}_k\}_1^{O_k}$ contains latent representations inputs to an operation, $\mathbf{P}_k$, and bit flip patterns, $\mathbf{F}_k$, that can produce $\boldsymbol{\delta_k}$ at the output of an operation when $\mathbf{x}_k$ is being computed.

We can determine $\mathbf{P}_k$ and $\mathbf{F}_k$ by analyzing the latent space of $F_k(\cdot)$ under $\mathbf{x}_k$ and $\boldsymbol{\delta}_k$, the operation reduced perturbation. Let $\widehat{\mathbf{P}}_k$ be the latent space representation immediately preceding the layers targeted by the algorithm and $\widetilde{\mathbf{P}}_k = op(\widehat{\mathbf{P}}_k)$, those immediately following them, under the

key sample, $\mathbf{x}_k$. Then, we can define $\mathbf{P}_k$ by masking the inputs to the target $op \in Ops$ using the Reduced Selection Mask, $\mathbf{R}$, previously found.

$$\mathbf{P}_k = \mathbf{R} \odot \widehat{\mathbf{P}}_k \tag{4.20}$$

Further, we understanding that we want $\mathbf{F}_k$ to be a mask such that $\mathbf{F}_k \oplus [\mathbf{R} \odot \widetilde{\mathbf{P}}_k]_b = [\mathbf{R} \odot \widetilde{\mathbf{P}}_k + \boldsymbol{\delta}_k]_b$. Thus, we can define $\mathbf{F}_k$ in terms of $\widetilde{\mathbf{P}}_k$ and $\boldsymbol{\delta}_k$ as:

$$\mathbf{F}_k = [\mathbf{R} \odot \widetilde{\mathbf{P}}_k + \boldsymbol{\delta}_k]_b \oplus [\mathbf{R} \odot \widetilde{\mathbf{P}}_k]_b. \tag{4.21}$$

Here $\oplus$ is the exclusive-or function. We use the notation $[\cdot]_b$ to denote the use of a binary representation of a value.

Mathematically, we can use $\mathbf{P}_k$ to define a trigger function for the operation, such that:

$$\tau_{op} = comp([\mathbf{P}_{t,op}]_b, [\widehat{\mathbf{P}}_{t,op}]_b) \tag{4.22}$$

where $[\widehat{\mathbf{P}}_{t,op}]_b$ is the binary representation of the input of op when a test input, $\mathbf{x}_t$, is computed by the model. Here $comp(\cdot,\cdot)$ is an operation that returns 1 if the binary representations of its inputs are the same and 0 otherwise. We can easily translate this function into a simple combinational circuit that activates a *trigger signal*, τ, when it detects a latent representation that matches that of $\mathbf{P}_k$.

The trigger signal can then be fed as input into a second circuit which produces the desired perturbation by flipping bits on the output. We define a mathematical representation of this perturbation functionality for op, using the flip patterns, $\mathbf{F}_k$.

$$pert(\mathbf{F}_k, \widehat{\mathbf{P}}_{t,op}), \tau_{op}) = \begin{cases} [op(\widehat{\mathbf{P}}_{t,op})]_b \oplus \mathbf{F}_k, & \text{when } \tau_{op} = 1, \\ [op(\widehat{\mathbf{P}}_{t,op})]_b, & \text{otherwise.} \end{cases} \tag{4.23}$$

This functionality is easily migrated into a minimal combinational circuit which, when embedded into a functional block, works in conjunction with the previously described modification to produce the watermark perturbation on op. Using this method, we can generate modifications that, when embedded into the hardware, produce the desired watermark signature under $\mathbf{x}_k$.

Algorithm 6 Block Constrained Perturbations

Require: $L(\cdot)$, $\mathbf{H}$, $F_k(\cdot)$, $\mathbf{x}_k$, $\mathbf{y}_k$
Hyperparameters: $\epsilon_{\boldsymbol{\delta}}$, $\epsilon_{\mathbf{B}}$, ϵ_{ψ}, $T_{\boldsymbol{\delta}}$, T_{ψ}, $T_{\mathbf{B}}$
Ensure: $F_k(\mathbf{x}_k) \neq \mathbf{y}_k$
$\mathbf{B} = \mathbf{1}$; $\boldsymbol{\delta} = \mathbf{0}$; $\psi^M = \psi = |\mathbf{B}|$
while $F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}_k) \neq \mathbf{y}_k$ **do**
  for $i \in [1, T_{\boldsymbol{\delta}}]$ **do**
    $\boldsymbol{\delta} = \boldsymbol{\delta} - \epsilon_{\boldsymbol{\delta}} \left[\frac{\partial L(F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}_k), \mathbf{y}_k)}{\partial \boldsymbol{\delta}} \right]$
  end for
  if $F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}_k) = \mathbf{y}_k$ **then**
    $\psi^{(i)} = \lceil \psi^{(i-1)} * \epsilon_{\psi} \rceil$; $\psi^M = \psi^{(i-1)}$;
  else
    $\psi^{(i)} = \lceil \psi^{(i-1)} + \frac{1}{T_{\psi}} * (\psi^M - \psi^{(i-1)}) \rceil$
  end if
  $\mathbf{Z}_1 = \mathbf{Z}_2 = \mathbf{Z}_3 = \mathbf{1}$; $\psi_0 = |\mathbf{B}|$
  for $i \in [1, T_{\mathbf{B}}]$ **do**
    $\mathbf{B} = \mathbf{B} - \epsilon_{\mathbf{B}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{B}} \right]$ {$\mathcal{L}$ is defined in Equation (4.25)}
    $\psi_n = \psi_{n-1} - \frac{1}{T_{\mathbf{B}}} * (|\mathbf{B}| - \psi)$ {Update the dual parameters using Equation (4.16)}
  end for
end while
$\widehat{\boldsymbol{\delta}}_k = \boldsymbol{\delta} \odot \mathbf{BH}$
return $\widehat{\boldsymbol{\delta}}_k$

4.2.2.6 Cardinality Constraint Optimization

The selection of ψ is critical to the strength of the watermark embedding as it contributes to determining the number of functional blocks targeted by the algorithm. DeepHardMark applies a brute force method of determining ψ by decreasing constant selections for ψ until the algorithm cannot find a solution. However, such strategies result in inferior solutions. In the DeepHardMark$^+$ algorithm, we improve the selection of this parameter by integrating it into the algorithm allowing it to settle into a more desirable solution.

We do this by initializing ψ to the largest constraint possible, $\psi = |\mathbf{B}|$, i.e., the total number of functional blocks in the hardware. Then, after updating $\mathbf{B}$ and $\boldsymbol{\delta}$, we verify that a valid solution was found by asserting that $F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}_k) = \mathbf{y}_k$. If the algorithm finds a valid solution, we record it as $\psi^M = \psi$. Then, tighten the constraint by setting $\psi = \lceil \psi * \epsilon_{\psi} \rceil$ where $\epsilon_{\psi} \in (0, 1)$ determines the rate of decrease. If a solution is not found, we instead relax the constraint by perturbing ψ towards ψ^M, the lowest observed value which produced a valid solution, using $\psi = \lceil \psi + \frac{1}{T_{\psi}} * (\psi^M - \psi) \rceil$. The cooling temperature, T_{ψ}, is a variable used to control how swiftly ψ returns to ψ^M. This framework allows us to dynamically decrease the cardinality constraint as the algorithm converges and then

relax the constraint when it becomes too difficult for a solution to be found.

The cardinality constraint, ψ, is used in Equation (4.4) as a target for the number of Functional blocks to be selected by $\mathbf{B}$. However, while solving for $\mathbf{B}$, we can further improve our control over the convergence rate by targeting a decreasing schedule of ψ_n converging to ψ. To accomplish this, we define the following rule for migrating towards ψ while optimizing $\mathbf{B}$.

$$\begin{aligned} \psi_0 &= |\mathbf{B}| \\ \psi_n &= \psi_{n-1} - \frac{1}{T_B} * (|\mathbf{B}| - \psi) \end{aligned} \tag{4.24}$$

T_B refers to the number of iterations used in solving for $\mathbf{B}$. We then rewrite Equation (4.15) with ψ_n.

$$\begin{aligned} \frac{\delta \mathcal{L}}{\delta \mathbf{B}} = \frac{\delta L(F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(\mathbf{x}_k), \mathbf{y}_k)}{\delta \mathbf{B}} + \rho_1(\mathbf{B} - \mathbf{S}_1) + \mathbf{Z}_1 \\ + \rho_2(\mathbf{B} - \mathbf{S}_2) + \mathbf{Z}_2 + [\rho_3(\mathbf{1}^T\mathbf{B} - \psi_n) + \mathbf{Z}_3]\mathbf{1}. \end{aligned} \tag{4.25}$$

This update produces a more elastic downward pressure on $\mathbf{B}$, which allows the algorithm to slowly settle into a valid solution over multiple iterations of the algorithm, often contributing to a better solution. Utilizing these techniques, we can produce the optimized algorithm denoted as DeepHardMark$^+$ seen in Algorithm 6.

4.3 Experimental Evaluations

4.3.1 Experimental Setup

To demonstrate the effectiveness of the proposed methodology, we provide a comprehensive evaluation of our methodology when embedding watermarks into hardware using various well-known image classification and natural language benchmark architectures as the key DNN.

4.3.1.1 Datasets

In this work, we utilized three datasets, Cifar10 [86], Cifar100, and ImageNet [37]. These are three image classification datasets commonly used in deep learning research. This section presents the details of these datasets. All of these datasets have been made publicly available by the original source. The Cifar10 [86] dataset is a small dataset containing $60,000$ 32×32 color images. These images are evenly drawn from 10 distinct classes. We utilize the standard test/training split deter-

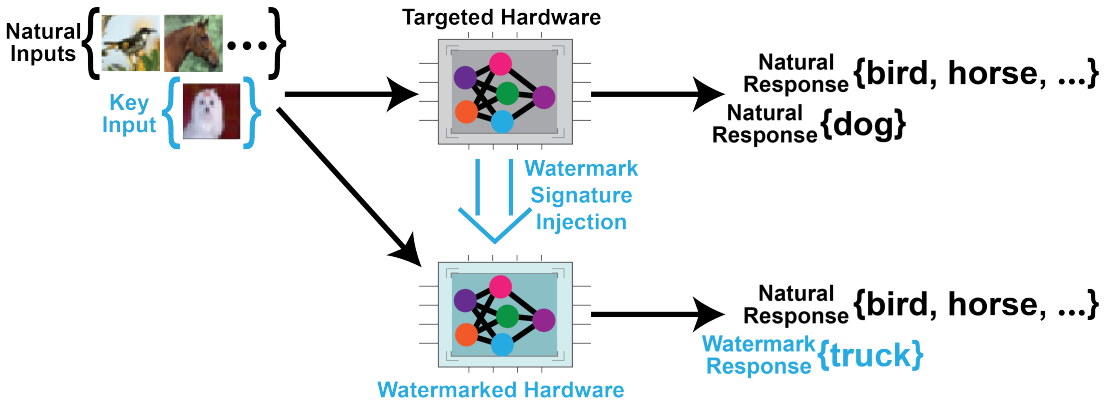

Figure 4.4: To verify the efficacy of DeepHardMark, we embed watermark modifications into two deep learning hardware accelerators. Through the proposed methodology we are able to embed the watermark signature, which alters the system's functionality when computing a target Key Sample on a corresponding Key DNN. It does this while preserving the functionality of the hardware both on the Key DNN and other models executed on the device. We do this through minimal hardware modifications targeted to small subset of the hardware's computational blocks.

mined by the source, which selects $1,000$ random images from each class as the testing set. The Cifar100 dataset is similar to the Cifar10 dataset but provides a more difficult image classification task. This dataset likewise contains $60,000$ 32×32 color images split into $10,000$ testing and $50,000$ training images. Unlike the Cifar10 dataset, the Cifar100 dataset is drawn from 100 distinct classes. The ImageNet [37] dataset is a large-scale image classification dataset composed of over 10 million images covering over $1,000$. Our images are preprocessed to a resolution of 224×224 and normalized according to the specifications of the target model. We utilize the standard validation split containing $50,000$ images drawn evenly from each class for testing.

We also use the IMDB and GLUE-SST2 sentiment analysis datasets. The IMDb dataset contains $50,000$ text reviews from popular movie titles with a 50/50 train/test split. The IMDb dataset is refreshed daily and publically available to the IMDb customers for personal and commercial use. Each review in the dataset is a label with a classification specifying whether the review is positive or negative towards the movie reviewed. The General Language Understanding Evaluation (GLUE) benchmark is a dataset covering various sub-tasks in natural language processing. Among these sub-tasks is the Stanford Sentiment Treebank (GLUE-SST2). This sub-task contains $68,000$ training and $1,000$ validation text reviews for popular movie titles and an associated positive/negative label. Both tasks are considered sentence-level two-way class split, i.e., binary sentience classification tasks.

4.3.1.2 Models

In our experimental evaluations, we utilize a broad range of models, including variations of the ResNet [67], VGG [139], DenseNet [71], and ViT [43] models. We utilize pre-trained models whenever possible. Specifically, for the ImageNet classifiers, we utilize the open-source models provided by the TIMM python library [159]. This library provides a standardized model organization providing a consistent interface allowing us to extend our algorithm from model to model efficiently. We utilize the ResNet 18, ResNet 34, and ResNet 50 models, frequently used convolutional neural networks. We evaluated the top-5 accuracy of these models to be 89.08%, 92.92%, and 93.86%, respectively. We also utilize VGG11, VGG13, VGG16, and DenseNet101 models trained to an accuracy of 91.95%, 94.03%, 93.70%, and 93.30%, respectively. The ViT-16-224 and Vit-32-224 models are transformer-based models that have recently received significant attention as state-of-the-art image classifiers achieving accuracies of 97.66 and 96.82.

We also utilize ResNet 18 and ResNet 34 to classify the Cifar10 and Cifar100 datasets. However, due to the limited availability of open-source models, we train the classifiers for these datasets. We utilize two open-source resource training methodologies to ensure the quality of these models. Using this framework, we train the ResNet 18 Cifar10 classifier to an accuracy of 93.02% and the ResNet 34 classifier to an accuracy of 93.34% [81]. Similarly, we also train a ResNet 18 and ResNet 34 Cifar100 classifier to a top-5 accuracy of 87.66% and 88.54% [157].

For natural language Processors, we use DistilBERT [132] and RoBERTa [102] sentiment analysis models.

4.3.1.3 Hardware Platforms

We implemented a target hardware centered around a Matrix Multiply Unit (MMU) composed of a 32×32 MAC array, similar to the TPU architecture. We composed $\mathbf{H}$ for all of the experiments using this hardware architecture which utilizes a weight stationary hardware mapping scheme. For our hardware experiments, we implement this design in Verilog on an Ultrascale+ Kintex using the Xilinx Vivado and an ASIC design using Synopsys Design Compiler by mapping to a 32nm technology node. We embed the watermark modifications into the design to determine their cost from the hardware perspective.

4.3.1.4 Evaluation Metrics

We evaluate the embedded hardware watermarks from both the algorithm and hardware perspectives. To do this, we utilize various metrics that help quantify different aspects of the embedded watermark's efficacy. To help in this evaluation, we define the following metrics.

- **Embedding Success Rate** (ESR) quantifies the success rate of producing modifications that can alter the key DNN's functionality on the modified hardware. Formally, we define this metric as:

$$ESR = \frac{1}{K}\sum_{k=1}^{K}(F_k^{\boldsymbol{\delta}_k}(\mathbf{x}_k) == \mathbf{y}_k) \times 100\%. \tag{4.26}$$

  K is the number of key samples used in the evaluation.

- **Accuracy Difference** (ΔAcc) measures the effect of embedded modifications on the key DNN's functionality on a subset of its natural inputs. We calculate this value with the following equation over a set of validation data.

$$\Delta\text{Acc}(F_k(\cdot)) = |\text{Acc}(F_k^{\boldsymbol{\delta}_k}(\cdot)) - \text{Acc}(F_k(\cdot))|. \tag{4.27}$$

  This metric is used to evaluate the scenario in which the key DNN is executed on the modified hardware, but the key sample is not present.

- **Fidelity Difference** (ΔFid) measures the fidelity in the hardware's algorithmic functionality. We quantify this characteristic using the following:

$$\Delta\text{Fid}(F(\cdot)) = |\text{Acc}(F^{\boldsymbol{\delta}_k}(\cdot)) - \text{Acc}(F(\cdot))|. \tag{4.28}$$

  This metric evaluates the modified hardware's functionality on alternative benchmark models $F(\cdot)$ that were not used as $F_k(\cdot)$ on a validation dataset.

- **Triggering Ratio** (T_{ratio}) is a metric used in quantifying how active the modifications embedded in a design are. The triggering ratio is defined as:

$$T_{ratio} = \frac{\#\text{ of times triggered}}{\#\text{ of evaluations}} \times 100\%. \tag{4.29}$$

  The more active the hardware modifications are in a circuit, the more likely it is for them

Dataset	Model (Acc%)	$\boldsymbol{\rho}_k\% \pm SD$	$ESR\% \pm SD$	ΔAcc% $\pm SD$	ΔFid% $\pm SD$
Cifar10	ResNet18 (93)	0.18 ± 0.09	100.0 ± 0.00	0.68 ± 0.14	0.12 ± 0.80
Cifar100	ResNet18 (77)	1.29 ± 0.86	100.0 ± 0.00	0.30 ± 0.42	0.25 ± 0.39
ImageNet	ResNet18 (89)	0.15 ± 0.07	100.0 ± 0.00	0.67 ± 0.47	0.68 ± 0.47

Table 4.1: Performance of the Proposed Hardware Watermarking on DNN Accelerators

to produce abnormal effects like increased power draw. Ideally, T_{ratio} should be as small as possible.

4.3.2 Experimental Results

4.3.2.1 Efficacy Analysis

In Table 4.1, we evaluate the efficacy of embedding watermarks by using the proposed framework and its impact on the system from both the algorithmic and hardware perspectives. It should be noted that in computing ΔFid, we calculate the metric for multiple benchmark DNNs and average the results. The breakdown of the individual results, along with the models T_{ratio}, for Cifar10 are shown in Table 4.2. The value of $\boldsymbol{\rho}_k\%$ represents the percentage of operations in the key DNN that is targeted for modification, which is quite small for all the models. As each of these operations needs to be represented in the hardware modifications and contribute to functional changes in the DNN, we observe that this value tends to correlate with the impact of the embedded modifications.

It can be seen from these results that the ESR of the proposed scheme is 100% for all the scenarios evaluated. This is possible because we can relax the carnality constraint, ψ, in Equation(4.4) until we can modify enough of the hardware blocks to ensure a solution is found. Our experimental results demonstrate that the overall impact of the modifications on both the hardware overhead and algorithmic functionality is minor. Note that the hardware performance is evaluated based on FPGA/ASIC accelerators. For example, we observe that both the ΔAcc and the average ΔFid are under 0.7% for all scenarios. Likewise, the embed watermark only increases the hardware overhead of the device by 1% for the ImageNet classifier. We can also conclude that the proposed methodology generalizes well to hardware intended for large-scale datasets.

Model	Acc%	T_{ratio}%	Δ Fid%
VGG11	91.95	0.67	0.206
VGG13	94.03	0.67	0.218
VGG16	93.70	0.75	0.262
VGG19	93.63	0.78	0.234
ResNet34	92.92	0.14	0.009
ResNet50	93.86	0.26	0.009
Dense121	93.30	0.17	0.019

Table 4.2: Impact on the Functional Fidelity.

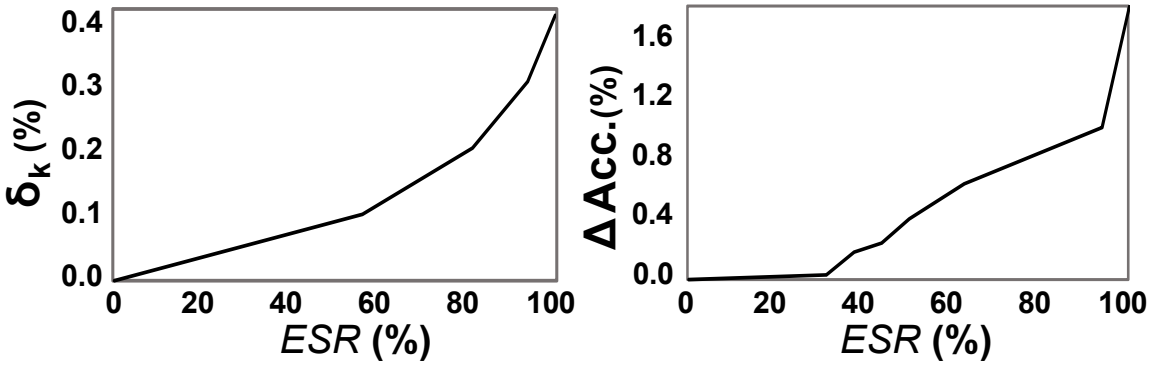

Figure 4.5: Functionality and Hardware Trade-offs

4.3.2.2 Hardware/Functionality Trade-offs

In the previous experiments, we ensured a 100% ESR by relaxing the limitation on the cardinality constraint, ψ. Now we study the relationship between ESR and the methodology's impact on the target hardware under smaller values of ψ. We plot ESR against ΔAcc and ESR verse $\boldsymbol{\delta}_k$%, the number of functional hardware blocks modified for the Cifar10 ResNet18 classifier, in Figure 4.5. These plots exhibit an obvious trade-off between ESR and the yield impact, in terms of both ΔAcc and $\boldsymbol{\delta}_k$%. Nevertheless, the overall modifications generated by the hardware watermark from both algorithmic and hardware perspectives are small. On the other hand, we can also effectively reduce such modifications if a smaller ESR is acceptable, as long as there is sufficient entropy for IP ownership verification. Both ΔAcc and $\boldsymbol{\delta}_k$% are halved if ESR can be relaxed.

4.3.2.3 Hardware Overhead

Finally, we evaluate the overhead required for embedding a watermark into a target DNN hardware accelerator. As we noted above, we use a target hardware with a 32×32 Matrix Multiply

Design	LUT	FF	DSP	Power (W)
Watermark-free	4427 (2%)	27808 (6.4%)	512 (28%)	0.592
Watermarked	4435 (2%)	27808 (6.4%)	512 (28%)	0.593
Overhead	0.18%	0%	0%	0.17%

Table 4.3: FPGA Hardware Overhead. Utilization is reported inside the parenthesis.

Unit(MMU) similar to [21]. We select a random modification set from the experiments above. We implement a combinational circuit that can embed the targeted functionality into the Verilog design. The results of the hardware overhead on Ultrascale+ Kintex FPGA are summarized in Table 4.3. It can be seen that the magnitude of hardware modification is minimal. For instance, there is only a 0.18% increase in the number of LUTs used, while the utilization for FF and DSP remains the same. The power overhead is also only 0.17%, which further verifies the transparency of the proposed hardware watermarking method. In addition, we present the results from ASIC implementation in Table 4.4, which is based on TinyTPU [137], a small-scale version of Google's TPU processor. We also extend the FPGA MMU design to ASIC. We can directly apply the watermark modifications to these designs with little complication. We also observe very little overhead in this scenario, with only a 0.054% increase in area and a 0.038% increase in power consumption.

	Area	Cells	Power	Time
TinyTPU	0.144%	0.119%	0.169%	0.00%
MMU	0.054%	0.058%	0.039%	0.00%

Table 4.4: ASIC Hardware Overhead: TinyTPU.

4.3.3 Evaluating DeepHardMark$^+$

4.3.3.1 Broad Evaluation of DeepHardMark$^+$

We continue our experimental evaluations by presenting the effectiveness of the DeepHardMark$^+$ algorithm. The improvement in computational efficiency of these optimizations allows us to easily extend our results to larger, more complex models and hardware architectures demonstrating a significant benefit. We target a larger hardware design for these experiments containing a 128×128 MMU and show that we can embed watermarks while targeting fewer operations in the key DNN. We utilize the procedures discussed in Section 4.3.1.

In Table 4.3.3.1, we present the results of our experimental evaluations on the improved

Dataset	Model	Acc%	$ESR\%$	$\lvert\delta_k\rvert\% \pm SD$	ΔAcc% $\pm SD$	ΔFid% $\pm SD$
Cifar10	ResNet18	93.02	100	0.001 ± 0.001	0.00 ± 0.00	0.00 ± 0.00
	ResNet34	93.34	99	0.001 ± 0.001	0.00 ± 0.00	0.00 ± 0.00
Cifar100	ResNet18	87.66	100	0.004 ± 0.003	0.00 ± 0.00	0.00 ± 0.00
	ResNet34	88.54	96	0.003 ± 0.001	0.00 ± 0.00	0.00 ± 0.00
ImageNet	ResNet18	89.08	92	0.017 ± 0.015	0.11 ± 0.08	0.00 ± 0.00
	ResNet50	92.86	100	0.004 ± 0.009	0.16 ± 0.38	0.00 ± 0.00
	VGG11	87.40	100	0.001 ± 0.002	0.23 ± 0.62	0.06 ± 0.15
	VGG16	96.88	100	0.004 ± 0.010	0.15 ± 0.42	0.06 ± 0.21
	WideResNet50	94.08	97	0.021 ± 0.046	0.13 ± 0.29	0.12 ± 0.28
	EfficientNetB2	95.31	98	0.020 ± 0.048	0.10 ± 0.27	0.10 ± 0.27

Table 4.5: Evaluating the Effectiveness and Impact of DeepHardMark$^+$ Watermark Modifications in Image Classification

methodology on an array of image classifiers, including ResNet50, WideResNet, and EfficientNetB2. We observe that the enhanced algorithm can successfully target these large CNN image classifiers with a high success rate, i.e., $> 90\%$. It also produces watermark embeddings that target a smaller percentage of the model's operations than the base algorithm while being less likely to affect the key DNN's accuracy and fidelity negatively. We further highlight the minor impact of the algorithm with the trigger ratio, T_{ratio}, which is very small for all key DNNs.

4.3.3.2 Extension to Alternative Deep Learning Scenarios

In Table 4.6, we directly extend our experimental evaluations to state-of-the-art transformer-based classifiers and Natural Language Processors. We assume a 128×128 MMU-based hardware design for this evaluation. Similarly to the previous experiments, we produce 100 watermark embeddings generated with randomly selected testing inputs targeting the ReLU functional blocks. We utilize our simulation framework to analyze the impact of the watermark embedding on ViT and Swin ImageNet classifiers, as well as DistilBERT and RoBERTa sentiment analysis models. We present the ESR and $|\boldsymbol{\delta}_k|\%$ observed in Table 4.6. Despite the significant differences between transform models and convolutional neural networks, DeepHardMark$^+$ achieves similar levels of performance on ViT and Swin models to their CNN counterparts. We observe a drop in the embedding success rate as we transition to the natural language processing setting. However, the ESR remains above 65% in this setting. Despite this, we note that the embedding DeepHardMark$^+$ produces highly efficient watermark embedding for this domain with less than 0.001% of the model operations being

targeted. The high success rates and low impact of these results demonstrate that our methodology can easily extend into other application domains with minor optimizations for the various target scenarios.

Dataset	Model	Acc%	ESR	$\lvert\delta_k\rvert\% \pm SD$
ImageNet	ViT-16-224	97.66	100	0.028 ± 0.036
	ViT-32-224	96.82	96	0.020 ± 0.021
	Swin-b-224	97.34	83	0.004 ± 0.002
IMDb	DistilBERT	92.79	100	0.001 ± 0.001
	RoBERTa	94.66	66	0.001 ± 0.001
GLUE-SST2	DistilBERT	98.85	70	0.001 ± 0.001
	RoBERTa	92.55	70	0.001 ± 0.001

Table 4.6: Evaluating the Effectiveness of DeepHardMark$^+$ in Transformers and Natural Language Processing Models

Table 4.7: Hardware Utilization of DeepHardMark$^+$ in FPGA Designs

	TinyTPU			
	LUT	DSP	Reg	Power (mW)
Watermark-free	13107	8040	256	0.531
Watermarked	13134	8044	256	0.528
Overhead	0.21%	$> 0.01\%$	0%	0.56%
	MMU			
	LUT	DSP	Reg	Power (mW)
Watermark-free	19096	36767	384	0.529
Watermarked	19133	36768	384	0.529
Overhead	0.19%	$> 0.01\%$	0%	0%

4.3.3.3 Hardware Evaluation of DeepHardMark$^+$

We then evaluated the hardware impact of the DeepHardMark$^+$ watermark embedding. For this evaluation, we implement a 32×32 version of the MMU and TinyTPU hardware accelerators in Quartus. We generate a watermark embedding for the ResNet18 Cifar10 classifier and embed modification which produces the watermark in the design. We synthesize the watermarked and watermark-free designs for a Cyclone V FPGA and determine its resource utilization in terms of Look Up Tables (LUT), Registers (Reg), and Digital Signal Processors (DSP) as well as its estimated

Table 4.8: Hardware Overhead of DeepHardMark$^+$ in ASIC Designs

	MMU			
	Area	Cells	Power	Time
TinyTPU	0.042%	0.048%	0.09%	0.01%
MMU	0.013%	0.011%	0.01%	0.04%

power draw. For the power consumption estimation, we assumed an I/O toggle rate of 12.5%. We present the recorded FPGA utilization in Table 4.7.

This process is repeated for ASIC design using a Synopsys 32nm technology node. We utilize the same watermark embedding and hardware designs. For the ASIC design implementations, we determine the area and cell count. Power consumption and propagation delay of the system. The recorded estimations of these characteristics are presented in Table 4.8.

From these results, we observe that the impact of the watermark modifications on the hardware overhead is minimal. This finding is consistent with prior evaluations of DeepHardMark. For instance, the increase in register and DSP utilization in the FPGA implementations and the power consumption and propagation delay of the ASIC designs are all less than 0.01%. Further, in the ASIC designs, the largest impact is the increase in the required cells, which requires only a 0.045% expansion. Comparing these results with those presented in Section 4.3.1, we demonstrate that the DeepHardMark$^+$ algorithm can produce hardware modifications with a decreased impact on the hardware overhead of the target design.

4.4 Conclusions

4.4.1 Summary

In this work, we proposed a watermark embedding methodology for Deep learning hardware accelerators. This framework links a hardware design with a key DNN and key samples provided by the hardware owner. In this scenario, modifications are introduced to the hardware that detects when the device is computing the target key DNN and samples and alters the behavior of the model by perturbing internal operations. The behavioral change is not detected under benign scenarios and so can be effectively used to identify a hardware platform. This scenario is similar to software deep learning watermark embedding schemes, but attaching the watermark signature directly to the

hardware can provide protection for the hardware platform.

Further, our proposed algorithm utilizes a novel sparse optimization algorithm that effectively minimizes the magnitude of the hardware modifications embedded and reduces their impact on the hardware's functionality. Due to this sophisticated algorithm, we are able to develop an algorithm that is very effective in embedding the watermark signature with minimal impact on the design itself. We are further able to optimize this algorithm with improvements that search the defense hyper-parameters to produce consistent, high-quality watermark embedding modifications. Our experimental results have demonstrated the efficacy of the proposed scheme to preserve the intended functionality, produce a minimal impact on the consumption of hardware resources, and effectively produce the desired output response.

Chapter 5

Compromising Embedded Deep Learning Based Security Systems

This work was presented in Symposium Series on Computational Intelligence (SSCI) 2021.

5.1 Security of Deep Learning Based Security Systems

Due to the significant success seen by deep learning in recent years, deep learning is being deployed in a wide array of application domains. Significantly, deep learning is being used in edge, IoT, and embedded systems where traditional computing approaches have been seen to be limited. This transition has led to strong development in both the deep learning and application perspectives. From the application perspective, novel machine learning systems are enabling the development of cutting-edge technologies such as autonomous driving, interconnected smart home systems, and intrusion detection systems. These are all very powerful systems that have, and will continue to have, an out-sized effect on modern society.

One major of the tasks for deep learning in embedded and IoT applications is as the center of security systems. Such security solutions are useful, especially in resource-constrained or dynamic settings where traditional approaches fail. Developing such systems often requires a degree of hardware/software co-design in which highly efficient hardware is paired and optimized alongside deep learning models designed to fully utilize that hardware without a loss in functionality. Un-

fortunately, in such settings, resource limitations often limit external security solutions. As such, since deep learning is known to be susceptible to adversarial attacks can easily become a major vulnerability in the system despite its intent to defend the system.

In this chapter, we explore the vulnerability of deep learning in resources constrained settings by compromising a popular Network Intrusion Detection System (NIDS), Kitsune, against adversarial examples. This system is a very low-powered auto-encoder based NIDS designed to be deployed to IoT hardware. As this system is often deployed to resource-constrained settings, it is unlikely that it would be paired with systems designed to protect it from various deep learning based attacks. We demonstrate that an adversary could create adversarial network traffic that fools the system into classifying benign traffic as malicious or malicious traffic as benign.

5.1.1 Network Intrusion Detection Systems

In recent years, the increasing frequency and size of cyber-attacks in recent years [72] have made network intrusion detection systems (NIDS) a critical component in network security. An example of a network intrusion detection system is shown in Figure 5.1. The intrusion detection system essentially acts as a gatekeeper at the target node, which activates a firewall or alerts a host device when malicious network traffic is detected. Unfortunately, while these systems can effectively defend the entry point, much of the network remains unprotected. In other words, attacks that remain internal to the network are often difficult to detect by the traditional intrusion detection systems [110].

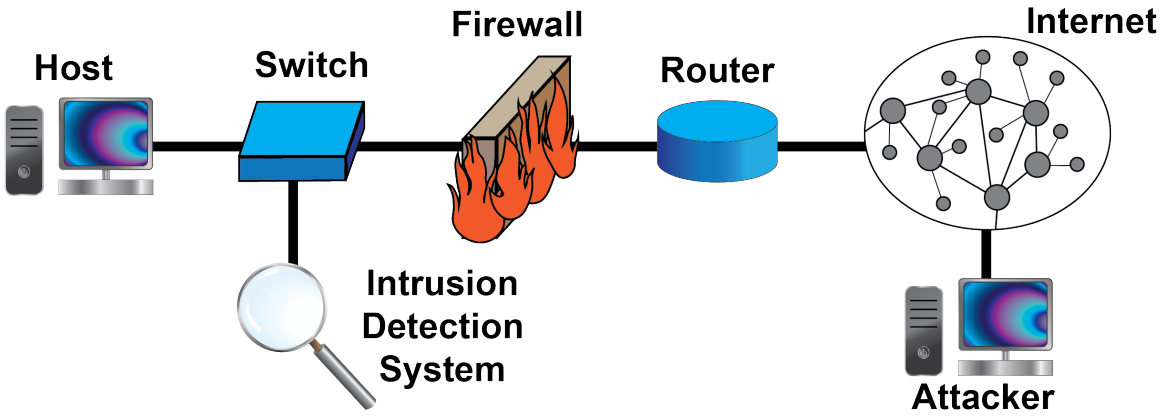

Figure 5.1: An intrusion detection system positioned to defend a host device from abnormal network traffic.

Deploying an intrusion detection system at multiple nodes distributed throughout the network can fill this hole to further secure networks. However, a significant drawback of the traditional rule-based approach is that each intrusion detection system must be explicitly programmed to follow a set of rules. This process also generates potentially long lists of rules that need to be stored locally to access intrusion detection systems. Furthermore, any changes in a network node might potentially lead to an update for the entire network. To this end, DL-NIDS have the potential to overcome this weakness as they can generalize the defense by capturing the distribution of typical network traffic instead of being explicitly programmed [110, 140, 76, 5]. In addition, these methods do not require large lookup tables, which could also reduce the implementation cost.

5.1.2 Adversarial Example Generation

A major focus of adversarial deep learning is the adversarial example generation, which attempts to find input samples by slightly perturbing the original benign data to yield different classifications. Formally, the adversarial example generation process can be expressed by [18]:

$$\begin{aligned} \text{minimize} \quad & \mathcal{D}(\vec{x}, \vec{x} + \vec{\delta}) \\ \text{such that} \quad & \mathcal{C}(\vec{x} + \vec{\delta}, \vec{t}) \\ & \vec{x} + \vec{\delta} \in \mathbb{X} \end{aligned} \tag{5.1}$$

Where $\vec{x}$ is the model's original primary input, $\vec{\delta}$ is a perturbation on $\vec{x}$ to achieve the desired adversarial behavior, and $\mathbb{X}$ defines a bounded region of the valid input values. $\mathcal{D}(\cdot)$ is a distance metric that limits δ, while $\mathcal{C}(\cdot)$ is a constraint that defines the goal of the attack. Two commonly used constraint functions are $F(\vec{x}) = \vec{t}$ and $F(\vec{x}) \neq \vec{t}$. The first defines a targeted attack in which the adversarial goal is to force the network output, $F(\vec{x})$, to a specific output, $\vec{t}$. The second defines the untargeted scenario where the adversarial goal is for the network to produce any output except $\vec{t}$. The choice of $\mathcal{D}(\cdot)$ also greatly affects the outcome of the attack. In the existing works, L_P norms (i.e., L_0, L_1, L_2, and L_∞) are often used due to their mathematical significance and correlation with perceptual distance in the image or video recognition. Recently, new distance metrics are being explored with recent works such as spatially transformed adversarial examples [164].

Many algorithms for generating adversarial examples utilizing various $C(\cdot)$, $D(\cdot)$, and optimization approaches have been developed in the literature. For example, one of the earliest adver-

sarial example algorithms, the Fast Gradient Sign Method (FGSM), perturbs every element of the input in the direction of its gradient by a fixed size [59]. While this method produced quick results, the Basic Iterative Method (BIM) can significantly decrease the perturbation, requiring a longer time to run [45]. Furthermore, adversarial example generation algorithms continue to grow more sophisticated as novel attacks build on the foundation of existing works. An example of this is the elastic net method (ENM) which adds an L_1 regularization term and the iterative shrinkage-thresholding algorithm to Carlini and Wagner's attack [24]. Moreover, adversarial examples are expanding out from image processing into alternate fields where they continue to inhibit the functionality of deep learning models [19, 73, 83]. The effort to draw researcher awareness to the subject has even led to the generation of competitions in which contestants attempt to produce and defend neural networks from this adversarial example [15, 87].

5.1.3 Robustness against Adversarial Examples

Some researchers believe that the vulnerability of deep learning models to adversarial examples is evidence of a pervasive lack of robustness rather than simply an inability to secure this models [52, 138, 56]. As such, defenses attempt to bolster the deep learning model's robustness by using either reactive or proactive methods [169]. Defensive distillation and adversarial training are two proactive defenses which improve a neural network's robustness by retraining the network weights to smooth the classification space [105, 45]. A recent example of a reactive defense is, PixelDefend, which attempts to perturb adversarial example input back to the region of inputs space that is correctly handled by the network [168].

When deep learning is powering security applications, the robustness of the model is even more critical. The field of malware classification is a prime example, as deep learning models have been shown to perform superbly in this area in multiple implementations and scenarios [60, 116, 115, 51]. Unfortunately, when adversarial examples are presented to these systems, the lack of robustness in the deep learning model often allows an attacker to bypass this security measures [91, 82]. Despite this vulnerability, deep learning is a prime candidate for security implementations when traditional defenses' resource demands or static nature inhibit their practicality. Thus, as deep learning continues to develop into network intrusion detection, the robustness of such systems should be thoroughly studied. To this end, researchers are continuing to develop guidelines and frameworks to aid in ensuring the robustness of machine learning systems against adversarial manipulations [55,

17].

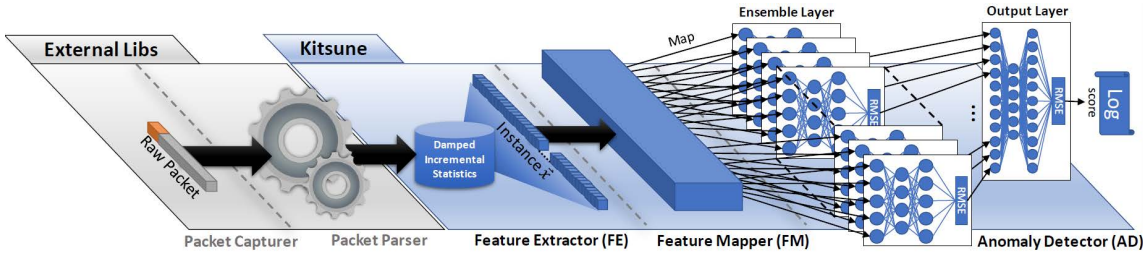

Figure 5.2: A graphical representation of Kitsune [110].

5.2 Evaluating the Network Intrusion Detection System

This section presents a brief overview of the network intrusion detection system. Then it analyzes Kitsune's deep learning model, KitNET, in more detail.

5.2.1 Kitsune Overview

The DL-NIDS, Kitsune, is composed of Packet Capturer, Packet Parser, Feature Extractor, Feature Mapper, and Anomaly Detector [110]. The Packet Capturer and Packet Parser are standard components of NIDS, which forward the parsed packet and meta-information (e.g., transmission channel, network jitter, capture time) to the Feature Extractor. Then, the Feature Extractor generates a vector of over 100 statistics which defines the packet and current state of the active channel. The Feature Mapper clusters these features into subsets fed into the Anomaly Detector, which houses the deep learning model, KitNET.

The Kitsune DL-NIDS is specifically targeted at being a lightweight intrusion detection system deployed on network switches in the IoT settings. Thus, each implementation of Kitsune should be tailored to the network node that it defends. This goal is achieved by using an unsupervised online learning approach that allows the DL-NIDS to dynamically update in response to the traffic at the target network node. The algorithm assumes that all real-time transmissions during the training stage are legitimate and thus learns a benign data distribution. For inference, it analyzes the incoming transmissions to determine if it resembles the learned distribution.

5.2.2 KitNET

KitNET, Kitsune's deep-learning backbone, consists of an ensemble layer and an output layer. The ensemble layer includes multiple autoencoders, each working on a single cluster of inputs provided by the Feature Mapper. The output scores of these autoencoders are normalized before being passed to an aggregate autoencoder in the output layer, whose score is used to assess the security of the network traffic data.

5.2.2.1 The Autoencoders

The fundamental building block of KitNET is an autoencoder. Autoencoders are a family of neural network that reduces an input down to a condensed base representation before reconstructing a tensor of the the same dimension as the input from that representation. The autoencoders in KitNET are trained to capture the properties of typical network traffic correctly by condensing the outputs of the Feature Extractor to its base representation, then reconstructing the original input. The number of hidden neurons inside an autoencoder is limited, so the network can learn a compact representation. When the network can reconstruct the input features it can be assumed that those features are representative of features the model has learned to reconstruct correctly, i.e. normal network traffic. Otherwise, we can consider the input features to be abnormal.

KitNet is composed of a two layers of autoencoders that are dynamically generated during training using a clustering algorithm to group similar features. The clustered features are feed through seperate auto encoders. The reconstructed features are then concatenated and feed through a final auto encoder. This hierarchical structure compartmentalizes information and helps KitNET learn the patterns of natural network traffic more quickly.

KitNET employs a root-mean-squared-error (RMSE) function on each autoencoder as the performance criteria. The score generated by each autoencoder block is given by:

$$s(\mathbf{x}) = RMSE(\mathbf{x}, F(\mathbf{x})) = \sqrt{\frac{\sum_{i=1}^{n}(\mathbf{x}_i - F(\mathbf{x})_i)^2}{n}}. \tag{5.2}$$

Where n is the number of inputs. Because the model was trained to reproduce instances from $\mathcal{X}$, a low score indicates the input resembles the normal distribution well. Usinging RMSE we can quantify the correctness with which the model has reconstructed its inputs. This function will thus function both as a loss function during training and an indicator to flag abnormal network traffic.

5.2.2.2 The Normalizers

Another component used by Kitsune is the normalizers, appearing both before entering KitNET and before the aggregate autoencoder. These normalizers implement the standard function:

$$norm(\mathbf{x}_i) = \frac{\mathbf{x}_i - min_i}{Max_i - min_i}. \tag{5.3}$$

Which linearly scales minimum and maximum input values to 0 and 1, respectively. In Kitsune's training, the value of Max_i and min_i respectively take on the maximum and minimum input values seen by the $\mathbf{x}_{i_{th}}$ element during training.

5.2.3 Classifying the Output

The primary output of KitNET is the RMSE score, S, produced by the aggregate autoencoder. It should be noted that the scores produced by KitNET are numerical values rather than a probability distribution or its logits, like in standard deep learning classifiers. Kitsune utilizes a classification scheme that triggers an alarm under the condition: $S \geq \phi\beta$, where ϕ is the highest value of S recorded during training and β is a constant used to find a trade-off between the number of false positives and negatives. The authors limit the value of β to be greater than or equal to 1.0 to assure a 100% training accuracy (i.e., all the training data are considered benign).

5.2.4 Targeting the System

The deep learning model, KitNET is simply a sub-systems of the larger Kitsune NIDS. As such, targeting the model requires passing network packets through the Packet Capturer, Packet Parser, Feature Extractor, and Feature Mapper. Applying an adversarial example generation algorithm on this model will result in perturbations on the model's input that should be translated into changes in the network traffic. This means backpropigating this information through the other sub components of Kitsune. Fortunately, most of these subsystems can be trivially reversed. The Feature Mapper simply maps features to the inputs of KitNET and reversing this mapping enables us to know which features should be perturbed. Similarly, the Packet Capturer, Packet Parser, and Feature Extractor simply work together to convert the packet properties and meta-information into values that can be process by KitNET. These are standard subsystems for an NIDS and so it should

also be possible for an adversary with an understanding of networks to generate packets which are decomposed into the target feature. For example, one of the major features extracted by this system is the packet size. If the adversarial example requires this feature to be increased the adversary can pad a package to meet this objective.

5.3 Experimental Evaluations

5.3.1 Experimental Setup

In this section, we briefly describe our experimental setup and the necessary modifications to the KitNET.

5.3.1.1 Implementing KitNET

In order to perform adversarial machine learning, the original C++ version of Kitsune was reproduced in TensorFlow [1]. The TensorFlow model was tested and evaluated similarly to the C++ implementation with an average deviation on the outputs of 5.71×10^{-7} from the original model. We then utilized the Cleverhans [120], an adversarial machine learning library that is produced and maintained by domain experts, to mount different adversarial example generation algorithms on the Kitsune. We also used the same Mirai dataset as in [110]. This dataset contains features describing network traffic captured from a Mirai botnet attack on an emulated IoT network. The dataset contains network traffic which can be classified into 10 different sub categories indicating an attack. However, in the case of KitNET, the objective is to simply classify the traffic into either malicious or benign network traffic.

5.3.1.2 Modifications to the Model

Our implementation of KitNET moves the classification mechanism into the model by adding a final layer at the output, as expressed in Equation 5.4.

$$C(\mathbf{x}) = \begin{bmatrix} \text{benign} \\ \\ \text{malicious} \end{bmatrix} = S(\mathbf{x}) \begin{bmatrix} 1 \\ \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ \\ 2T \end{bmatrix} \tag{5.4}$$

This allows the deep learning model to produce the classification result based on a threshold, T. Effectively, this alteration moves the original classification scheme into KitNET itself when $T = \phi\beta$, transforming the model from a regression model into a classifier.

As adversarial examples target deep learning models, we isolate KitNET from Kitsune when performing our attacks. In a real-world attack on Kitsune, the adversary must circumvent or surmount the Feature Extractor to induce perturbations on KitNET's input. However, understanding the Feature Extractor makes it feasible for the adversary to craft network traffics to generate essential features. Thus, in our experiments, we focus on evaluating the security of KitNET from the normalized feature space.

5.3.2 Evaluations from the Network Security Perspective

A DL-NIDS must be evaluated from both the network security and adversarial machine learning aspects to understand its defensive capabilities fully. In the domain of intrusion detection, the ability to distinguish malicious network traffics from benign traffics is the primary performance metric. In this section, we evaluate the classification accuracy of the Kitsune.

Kitsune's developers evaluate the DL-NIDS against a series of attacks in a variety of networks [110]. In our implementation, the accuracy of Kitsune is highly dependent on the threshold, T. This value defines the decision boundary, which makes it a critical parameter when deploying the model. We evaluate the KitNET by assuming that the threshold is not predefined but trained as an end-to-end deep learning system. In addition, this analysis also indicates how the threshold correlates with the perturbation required in adversarial machine learning.

To assess the performance of a given threshold value, we consider the following two metrics:

1. **False Positives:** The percentage of benign inputs that are incorrectly classified as malicious.
2. **False Negatives:** The percentage of malicious data that are incorrectly classified as benign.

On the one hand, the rate of false positives accounts for the reliability of a network. On the other hand, the rate of false negatives is closely associated with the intrusion detection system's effectiveness. Therefore, both rates should be minimized in an ideal situation. However, in the setting of Kitsune, the value of T acts as a trade-off between a false-positive rate and a false-negative rate.

We investigated the whole functional range of possible thresholds in this analysis, i.e., from the minimum score of 0 to 20, which leads to 100% false negatives on the given dataset. Figure 5.3(a) plots the two metrics as well as the accuracy of the DL-NIDS.

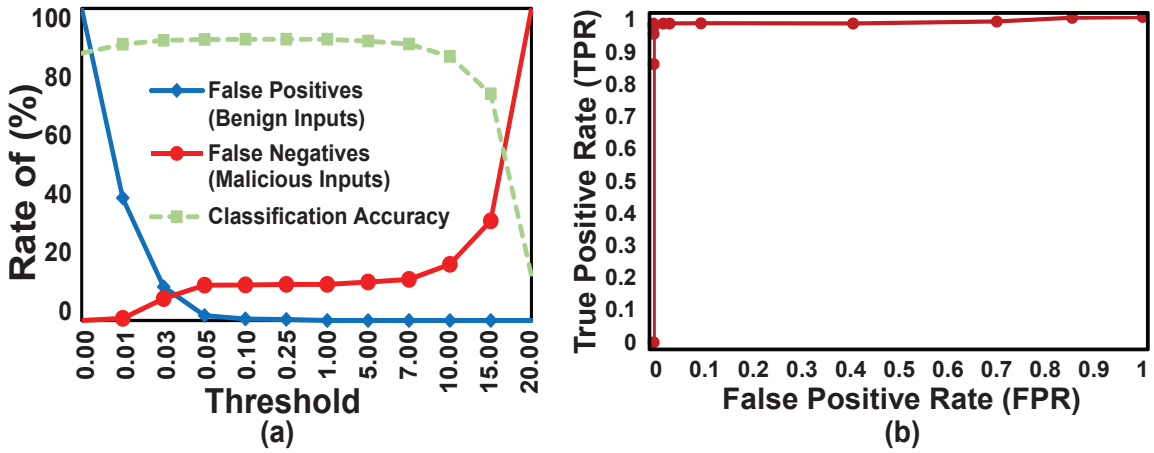

Figure 5.3: The percentage of misclassified benign and malicious inputs for chosen threshold values (a). A receiver operating characteristic (ROC) curve for Kitsune (b).

It can be seen that the rates of false positives and false negatives remain almost unchanged in the middle range. Furthermore, it can also be observed that if we want to minimize one of the rates, the other rate will increase significantly. Finally, the accuracy is also essentially unchanged for threshold values below 7, which can partially contribute to the imbalance of the dataset (i.e., most of the data belong to the benign class). Therefore, a threshold between 0.05 and 1 would be appropriate for this scheme. The effectiveness of Kitsune at separating the Mirai dataset is further demonstrated by the ROC curve in Figure 5.3(b).

5.3.3 Evaluations against Adversarial Machine Learning

This section continues the evaluation of Kitsune through an empirical analysis of its robustness against adversarial examples. Adversarial examples are particularly dangerous in such settings as they do not require direct access to the model internals to generate and so can be conducted even in remote settings.

5.3.3.1 Adversarial Example Generation Methods

Intelligent and adaptive adversaries will exploit the vulnerability of the machine learning models against novel DL-NIDS by using techniques such as adversarial examples and poisoning attacks. There are mainly two attacking objectives in adversarial machine learning, namely, integrity and availability violations. In this setting, integrity violations attempt to generate malicious traffic, which evades detection (produce a false negative), while availability violations attempt to make benign traffic appear malicious (produce a false positive) [6]. However, adversarial examples attempt to achieve a misclassification with perturbations as small as possible.

Another concern in performing these attacks is that the network data are fundamentally distinct from images, usually used in conventional adversarial machine learning. An adversarial example in the image domain is an image perceived to be the same by human observers but differently by the model. The L_P norm between the two images exemplifies visual distance and can be used as the distance metric. However, this definition fails in network security as observing network traffic at the bit-level is not generally practical. Therefore, the semantic understanding of these attacks in this setting is remarkably different.

One potential definition for adversarial examples in this scenario, which is facilitated by the architecture of Kitsune, is to use the extracted features generated by the model as an indication of the observable difference. Thus, we adopt the L_P distance on the feature space between the original input and the perturbed input as the distance metric. In particular, the L_0 norm correlates to altering a small number of extracted features, which might be a better metric than other L_P norms.

Many methods of generating adversarial examples have been developed. With each thriving in different settings, we attempt to generate a broad comparison of adversarial examples with different distance metrics in the network security domain. We evaluate the robustness of KitNET against the following algorithms:

- **Fast Gradient Sign Method (FGSM):** This method optimizes over the L_∞ norm (i.e., reduces the maximum perturbation on any input feature) by taking a single step to each element of $\vec{x}$ in the direction opposite the gradient [59].

- **Jacobian Base Saliency Map (JSMA):** This attack minimizes the L_0 norm by iteratively calculating a saliency map and then perturbing the feature that will have the highest effect [122]. The Jacobian Base Saliency Map algorithm has been a standard for generating

sparse adversarial example. Tis is a powerful attack in this setting as changing features in a packet may effect other features of the packet, and limiting the number of features to be perturbed significantly limits this risk.

- **Carlini and Wagner (C&W):** Carlini and Wagner's adversarial framework, as discussed earlier, can either minimize the L_2, L_0, or L_∞ distance metric [18]. Our experiments utilize the L_2 norm to reduce the Euclidean distance between the vectors through an iterative method.

- **Elastic Net Method (ENM):** Elastic net attacks are novel algorithms that limit the total absolute perturbation across the input space, i.e., the L_1 norm. ENM produces the adversarial examples by expanding an iterative L_2 attack with an L_1 regularizer [24].

5.3.3.2 Experimental Results

We conduct our experiments on both integrity and availability violations. Integrity violation attacks are performed on the benign inputs with a threshold of $s = 1.0$. The experimental results are presented in Table 5.1. For comparison between different algorithms, the common L_P distance metrics are all presented. Each attack was conducted on the same 1000 random benign samples from the dataset.

		L_P Distances			
Algorithm	Success (%)	L_0	L_1	L_2	L_∞
FGSM	100	100	108	10.8	1.8
JSMA	100	2.33	10.73	6.97	4.87
C&W	100	100	7.44	3.61	3.49
ENM	100	1.21	4.94	4.64	4.49

Table 5.1: Integrity Attacks on KitNET

Availability attacks are also performed using the same threshold of $s = 1.0$. 1000 input vectors that yield the closest output scores to the threshold were selected. The results are summarized in Table 5.2. As the normalizers were only trained on benign inputs, many malicious inputs would be normalized outside the typical range between 0 and 1.

		L_P Distances			
Algorithm	Success (%)	L_0	L_1	L_2	L_∞
FGSM	4	100	78.00	7.79	0.78
JSMA	0	–	–	–	–
C&W	100	100	22.00	8.50	5.61
ENM	100	8.74	21.7123	8.14	3.60

Table 5.2: Availability Attacks on KitNET

5.3.4 Analysis and Discussion

By comparing Table 5.1 and Table 5.2, it can be seen that the integrity attacks, in general, perform much better than the availability attacks. For instance, adversarial examples are rarely generated in the FGSM and JSMA availability attacks. Additionally, the perturbations produced by the availability attacks are all larger than their integrity counterparts. A potential cause for the difficulty is the disjoint nature between the benign and malicious input data, as exhibited by the clipping of the normalized inputs, in conjunction with a boundary decision (i.e., the threshold T) that is much closer to the benign input data.

Among these four methods, the earlier algorithms, i.e., the FGSM and JSMA, perform worse than the C&W and ENM attacks. As we mentioned above, especially in the availability attacks, the success rates of these attacks are significantly low. This result is expected since the more advanced iterative C&W and ENM algorithms can search a larger adversarial space than the FGSM and JSMA. However, as noted above the L_0 metric is significant in this setting as it quantifies the sparsity of the attack. When back propagating the attack through Kitsune's other sub components this could be a significant factor in the attack's success. As such, we specifically recognize the power of the ENM algorithm in this setting as it is the likely to produce more feasible real-world attacks. Further, we note that recent works have been striving to develop stronger sparse attacks for against deep learning systems and may provide significantly stronger attacks than those seen here.

A final observation is that ENM is very effective in these attacks. Even though this attack is optimized for the L_1 norm, its generated adversarial examples simultaneously yield minimal values for the other norms. Specifically, the L_0 perturbations produced were even better than those produced by JSMA. As stated above, the L_0 norm seems to be the most appropriate norm among these four L_p norms in the setting of network security, as it signifies altering a minimized number of extracted features from the network traffic. Thus, ENM can be implemented against

the Kitsune to generate adversarial examples to fool the detection system while requiring minimal perturbations.

We note that the above attacks were produced with an adaptive step size random search of the parameters of each method. In practice, adversaries may use such a naive approach to determine effective attack algorithms. Then, utilize more robust optimization algorithms, such as Bayesian or gradient descent optimization, with the indicated attack algorithms to produce a superior result.

5.3.5 Optimizing ENM

Since ENM has been demonstrated to be very successful in our experiments, we next focus on optimizing the ENM attack on Kitsune in our setting. The CleverHans implementation uses a simple gradient descent optimizer to minimize the function:

$$c \cdot \max\{F(\mathbf{x})_j - \mathbf{y}\ ,\ 0\} + \beta||\mathbf{x} - \mathbf{x_0}||_1 + ||\mathbf{x} - \mathbf{x_0}||_2 \tag{5.5}$$

where $F(\cdot)_j$ is the logit output of the target classifier, $\mathbf{y}$ is the target logit output (i.e., the output which produces the desired violation), and $\mathbf{x_0}$ is the original network input. It can be seen that there are two regularization parameters, c and β. These parameters determine the contribution of the different metrics to the attack algorithm. For example, a very large c effectively increases the attack's ability to converge to a successful attack. The large contribution of the constraint terms also potentially overshadows the distance metrics, effectively diminishing the attack's ability to minimize the perturbation. The focus of this optimization is to determine optimal regularization terms to produce effective attacks on KitNET.

The ENM algorithm has several other hyper-parameters, including the learning rate, maximum gradient descent steps, and targeted confidence level. These parameters are standard in adversarial example attacks; these parameters are set to the constant values of 0.05, 1000, and 0, respectively. An optimization scheme included in the ENM algorithm aids in producing optimal results by altering c. It decreases the parameter N-times, only retaining the successful attack, which produces the lowest perturbation. This feature is disabled by setting $N = 0$, ensuring that it does not alter optimization results. Therefore, the results of the optimization could be further improved by enabling this functionality.

The parameter, c, determines the contribution of the adversarial misclassification objective

at the cost of diminishing the two L_P normalization terms. Thus, it can be logically determined that the optimal value of c is that value that achieves the demanded success rate while remaining as small as possible. We evaluate a wide range of c values for $\beta = 1$, as shown in Figure 5.4. We find $c = 450$ optimal, which achieves a 100% success rate with a relatively small perturbation. It can also be observed from Figure 5.4 that the resultant L_1 distance does not directly correlate to the selection of c. We also tried to increase the value of c into the thousands; interestingly, the L_P distances still only changed very slightly.

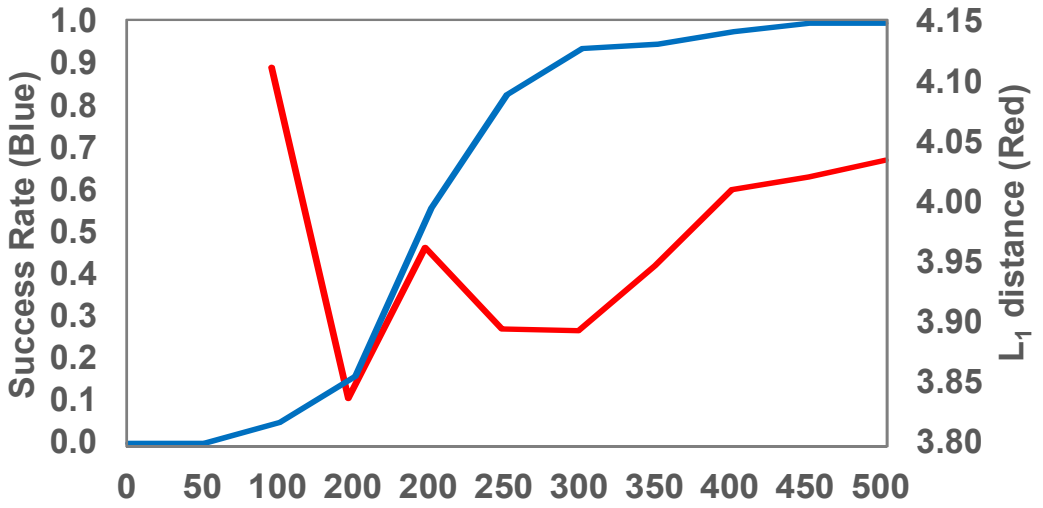

Figure 5.4: The success rate (blue) and average L_1-distance (red) of adversarial examples with respect to the regularization parameter, c, used for the attack.

On the other hand, the choice of β significantly affects the L_P distances. We now optimize the produced perturbation through varying the parameter β for $c = 450$. The results are summarized in Table 5.3. It can be seen that the success rate will drop as the increase of c, after the second term of Equation 5.5 begins to overpower the loss function associated with c.

It can be concluded that adversarial machine learning can be a real threat against DL-NIDS. Therefore, when moving intrusion detection towards the profound learning realm, it is critical to evaluate the security of a DL-NIDS against both adversarial attacks in the conventional network and the machine learning domains.

		L_P Distances			
β	Success (%)	L_0	L_1	L_2	L_∞
1×10^{-5}	100	96.61	5.9518	3.6378	3.5163
1×10^{-4}	100	78.46	5.7574	3.6388	3.516
1×10^{-3}	100	33.34	5.0577	3.6435	3.5268
1×10^{-2}	100	5.51	5.1722	3.7658	3.3129
1×10^{-1}	100	1.09	3.8624	3.7277	3.6450
1×10^{0}	100	1.01	4.0347	4.0158	4.0044
2×10^{1}	0.84	1.00	4.1350	4.1350	4.1350
5×10^{1}	0.08	1.00	4.2054	4.2054	4.2054
1×10^{2}	0	-	-	-	-

Table 5.3: The perturbations produced with respect to β.

5.4 Conclusions

This work has demonstrated the vulnerability of DL-NIDS to well-crafted attacks from the domain of adversarial machine learning. This vulnerability is present in deep learning-based systems even when the model achieves high accuracy for classifying benign and malicious network traffic. Therefore, researchers must take steps to verify the security of deep learning models in security-critical applications to ensure they do not impose additional risks; otherwise, it will defeat the purpose of using deep learning techniques to protect networks.

The existence of the Feature Extractor and the Packet Parser signifies that the Kitsune is at least partially utilizing domain knowledge of network traffic to generate its classification. Their applications strive to be as data-driven as possible to get the most benefit from deep learning models (i.e., they require little to no human knowledge to generate a function mapping). Thus, despite the current success of Kitsune and other DL-NIDS, as the field continues to develop, DL-NIDS will attempt directly converting network traffic to a classification utilizing end-to-end deep learning models. Furthermore, the human knowledge currently being used by modern DL-NIDS implies that to increase the probability of a successful attack, an adversary should understand this knowledge. Thus, as DL-NIDS continues to develop, evaluating the model against adversarial machine learning techniques becomes even more critical as attacks will no longer require this additional knowledge when targeting the system.

This work assumes that the adversary has direct knowledge of the target DL-NIDS, allowing them to directly generate inputs for the deep learning model. A potential drawback of this assump-

tion is that the perturbation requires to generate the adversarial examples does not directly correlate to the alteration on the network. Additionally, it does not account for the effect of that change on the network traffic on the host device. Despite this limitation, we believe that the attacks presented here could potentially constitute a strong attack against deep learning systems. Indeed, many of the adversarial examples generated for Kitsune could be trivially produced in the network traffic with general knowledge of the system. For example, one of the major contributing feature for fooling the model was the size of network packages. Often simply increasing the size of the adversarial packets by padding it with zeros was sufficient to conduct the attack.

Chapter 6

Conclusions and Furture Directions

6.1 Conclusions

This dissertation has presented pioneering works from the hardware perspective of adversarial deep learning. Our novel perspective begins by considering the possibility of embedding backdoors into deep learning models through modifications to its computational operations. We utilize this framework to embed hardware Trojans in deep learning accelerators which are able to produce backdoors in a deep learning system. We further extend this concept to defend deep learning hardware from piracy using hardware watermarks.

Our Trojan embedding methodology targets a deep learning model deployed to a hardware accelerator. By conceptualizing the model as a layered architecture of operations, we can determine where the operations will be executed in the hardware. The operational backdoor determines a perturbation that could be introduced to the layer of operations to produce targeted or untargeted output responses when computing on key inputs. These perturbations can be introduced through modifications in the hardware. Due to the reuse of hardware to compute multiple operations, modifications within a single hardware block can be condensed to decrease the required overhead. Using a modified JSMA adversarial example generation algorithm, we are able to embed Trojans in simple deep learning accelerators, which effectively compromise a target model with minimal additional hardware overhead.

We extend our work to evaluating the protection of deep learning hardware accelerators from piracy violations by embedding a watermark signature into the hardware design. These modifications

would work in conjunction with a key DNN and corresponding key samples known only by the owner. Under normal operating conditions, the hardware would function as intended according to the deep learning model deployed to the system. However, once the rightful owner uploads the key resources onto the hardware, the modification in the design will alter the functionality of the deep learning model to produce identifiable behavior. Our watermark embedding process utilizes a sophisticated algorithm to produce sparse perturbations constrained to a small number of hardware blocks. This enables us to produce very low-impact modifications with minimal impact of both the hardware's overhead and functionality.

In addition, to these, we also explore the implications of traditional deep learning attacks on deep learning in resource-constrained hardware. We demonstrate that such systems could become vulnerable components of a system due to developers not considering the adversarial perspective of deep learning technologies. Our evaluations find that it is possible to generate adversarial traffic packets that can be used to fool Kitsune, a popular low-power network intrusion detection system intended for tightly constrained hardware applications. Such systems are typically deployed to low-power hardware platforms, such as for IoT or SDN applications. The intended purpose of such systems would be to act as a security solution for the network, but such security systems could become a vulnerability of the system rather than a defense.

6.2 Related Works

Following our work on hardware, Trojans a number of works have attempted to develop various attack methodologies for deep learning hardware.

Another early work in this field developed a Trojan attack that generated a Malicious Category Recognition system (MCR) implemented alongside the first Multiplier and Adder Tree in the first layer of CNN FPGA accelerator [167]. When a trigger input is detected, the benign output is replaced with the Trojan response, effectively compromising the model. This work demonstrated that this scheme could be embedded with very low overhead, $< 0.01\%$, and work with very subtle input triggers to compromise the system. The authors also identify a trade-off in this scheme where the size of the Trojan trigger correlates with the impact of the Trojan in the original dataset. They further present some of the limitations of hardware Trojan stealth from a practical perspective and ways adversaries could bypass such schemes.

Some works have also explored hardware-based attacks which do not rely on Trojan injection. Bit-Flip attacks [9], for example, are able to compromise deep learning systems by using targeted Rowhammer attacks. The Rowhammer attack is able to alter weights in DRAM according to an adversarial perturbation. This enables the ability to conduct traditional perturbation-based attacks from the hardware domain, expanding the ability of adversaries to achieve the access required to conduct such attacks. Such attacks have been well studied, and various defensive techniques against them have been explored. Many works have been extending similar defensive techniques in this context [97].

The Layer-based Noise Injection Attack [117] introduces a *stealthy hardware intrinsic attack*. This work attempts to generate a Trojan, which can be introduced to the input of any layer in the model. This Trojan generates random and stealthy noise to random elements of a feature map. In conjunction with an input trigger, this hardware would perturb the model's feature maps and alter its predictions. Using their methodology, they are able to embed Trojans with an overhead of 4% extra LUTs, 5% extra DSPs, and 2% extra FFs to compromise the device. They were able to perform this attack while modifying less than 1% of the elements in most layers' feature maps. These findings largely align with our results on hardware Trojans targeting the design's computations and signify that intelligent adversaries can use multiple avenues of compromising deep learning models through the hardware in both a stealthy and effective manner.

Another subsequent work generates a Trojan embedding which utilizes a target DNN as the Trojan trigger rather than its inputs [92]. The authors propose that this methodology enables their Trojans to overcome trivial defensive mechanisms such as data pre-processing and encryption. In this setting, the trojan trigger monitors the interrupt signals and analyzes its cycle patterns to capture specific DNN models and activate the Trojan payload circuitry. This modifies only the functionality of the target DNN executed on the device. Similar to our proposed methodology, this methodology is able to produce an effective attack with $< 1\%$ hardware overhead and minimal deviations in power consumption.

Side-channel attacks are powerful attacks against computing systems. These attack attempt to capture the side-channel information from a target device during standard computation and determine from that information some secret knowledge from the device. This information can include power consumption, thermal dissipation, or electrical activity on ground lines. Such attacks have been frequently deployed against cryptography systems to reveal hidden keys to break encryption

algorithms. However, they can also be used to recover other information from a device. Recently, side-channel attacks have been deployed against deep learning systems to extract model information. Specifically, side-channel attacks have been used to extract binary neural networks from a device [46]. Further, works are continuing to develop stronger, more complex algorithms for using side-channel analysis to extract key information about deep learning systems [20].

6.3 Future Directions

The study of hardware Trojans has developed a wide variety of Trojan modifications. Each of these designs introduces different trade-offs and capabilities to the adversarial scenario. The implications of this variety of hardware Trojans on deep learning security are still unexplored. The stealth and effectiveness of a hardware Trojan can be greatly affected by the family of modifications injected. Exploring the broader implications of different Trojan attacks on deep learning systems is of critical importance to the field.

The stealth of the Trojan attack is one of the major concerns for such attacks. Embedding or altering combinational circuits into a design can be quite invasive both in terms of the impact on the hardware's functionality and overhead. However, modern Trojan attacks are able to mitigate these risks using powerful Trojaning techniques. Dopant-level Trojans, for example, are able to embed logical changes to hardware by altering the electrical properties of the circuits transistors [11]. Such Trojans only change the doping level of the circuit while leaving the layout of the transistors alone. This makes it very difficult to detect the attack without destroying the device. Therefore, a major potential direction for the field is to apply such advanced Trojaning to deep learning hardware to produce attacks with an undetectable effect on the Hardware design.

Another important component for the stealth of Trojans is that the functionality of the hardware design is unchanged under normal operation. Trojans composed of sequential logic circuits require a sequence of correct inputs rather than a single correct input to be activated. One promising line of research is to explore whether such hardware Trojan designs could improve the stealthiness of deep learning hardware Trojans further by taking advantage of this property.

In addition to the injecting of backdoors, hardware Trojans are also frequently used for other attacks. Many Trojan attacks have been developed to siphon information from a device through data leakage attacks. In many cases, the information contained in a deep learning model can be

privacy critical. Future works in this direction could pursue the possibility of leaking model weight, architectures, or user data through the usage of deep learning hardware Trojans.

For deep learning hardware watermarks to be most effective in identifying a hardware design, the hardware modifications should be well aligned to work with the algorithmic components. Within this context, a promising direction for future research in deep learning hardware watermarking is to develop signature embedding algorithms which simultaneously optimize the key samples and key DNN to work with the hardware modifications. Integrating these components into the watermark framework removes some of the burdens from the modifications allowing for modifications that could be less invasive for both the hardware's functionality and overhead. The major challenge in this direction is to maintain the reliability of the embedded watermarks to produce the signature response while ensuring the watermark signature is fully contained in the hardware allowing it to be identified.

An interesting future work along this line of research would evaluate the possibility of optimizing the input images to better align hardware modification within the target computational blocks allowing the designer to embed the watermark with fewer hardware modifications. From another perspective, one of the problems with embedding watermarks that can be activated by a large number of key samples is that the hardware modifications continually increase for each key sample used. An exciting direction for future works is to optimize a set of input key samples together such that the perturbations required by the hardware to activate each input is similar. By aligning the key samples in this way, they designer can minimize the magnitude of the hardware modifications introduced by crafting modifications that simultaneously target each input sample.

One novel line of research that is currently unexplored is the utilization of deep learning hardware to defend against privacy violation attacks. It is to simultaneously optimize the deep learning hardware for performance and defensive capabilities. For example, privacy violation attacks attempt to siphon some valuable information about a deep learning system's users or test examples through a deep learning system. Model inversion attacks are able to recreate examples from the training dataset with only access to a model it generated. Currently, state-of-the-art defenses for such attacks utilize differential privacy to obscure deep learning data. Such methods introduce randomness into a deep learning system such that while the overall trend of the model adheres to the target class, the truth of individual examples is obscure. This is an effective methodology for defending the privacy of individual elements of a dataset in many privacy-critical scenarios.

The introduction of noise to instill differential privacy in a deep learning system has tradi-

tionally been conducted from the software perspective. However, it is understood that many avenues for optimizing hardware introduce such noise to a system. Quantization, for example, reduces the resolution of data representations in a hardware system. This has many major benefits, from reducing data storage and transfer rates to reducing the computation time of model operations. It may be possible to simultaneously utilize the noise generated through quantization methods to develop hardware that helps preserve the privacy of models executed on the device naturally.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Yossi Adi et al. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium, Baltimore, MD, USA, August 15-17*, pages 1615–1631. USENIX Association, 2018.

[3] Ibrahim M Ahmed and Manar Younis Kashmoola. Threats on machine learning technique by data poisoning attack: A survey. In *International Conference on Advances in Cyber Security*, pages 586–600. Springer, 2021.

[4] Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *arXiv preprint arXiv:1801.00553*, 2018.

[5] Muder Almiani, Alia AbuGhazleh, Amer Al-Rahayfeh, Saleh Atiewi, and Abdul Razaque. Deep recurrent neural network for iot intrusion detection system. *Simulation Modelling Practice and Theory*, 101:102031, 2020.

[6] Giovanni Apruzzese, Michele Colajanni, Luca Ferretti, Alessandro Guido, and Mirco Marchetti. On the effectiveness of machine and deep learning for cyber security. In *2018 10th International Conference on Cyber Conflict (CyCon)*, pages 371–390. IEEE, 2018.

[7] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *International Conference on Machine Learning (ICML)*, 2018.

[8] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. In *Usenix Security*, 2021.

[9] Jiawang Bai, Baoyuan Wu, Yong Zhang, Yiming Li, Zhifeng Li, and Shu-Tao Xia. Targeted attack against deep neural networks via flipping limited weight bits. *arXiv preprint arXiv:2102.10496*, 2021.

[10] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *Proceedings of the ACM Symposium on Information, computer and communications security*, pages 16–25, 2006.

[11] Georg T Becker, Francesco Regazzoni, Christof Paar, and Wayne P Burleson. Stealthy dopant-level hardware trojans. In *Cryptographic Hardware and Embedded Systems-CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings 15*, pages 197–214. Springer, 2013.

[12] Shivam Bhasin and Francesco Regazzoni. A survey on hardware trojan detection techniques. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2021–2024. IEEE, 2015.

[13] Chandradeep Bhatt, Indrajeet Kumar, V Vijayakumar, Kamred Udham Singh, and Abhishek Kumar. The state of the art of deep learning models in medical science and their challenges. *Multimedia Systems*, 27(4):599–613, 2021.

[14] Battista Biggio, Blaine Nelson, and Pavel Laskov. Support vector machines under adversarial label noise. In *Asian conference on machine learning*, pages 97–112. PMLR, 2011.

[15] Wieland Brendel, Jonas Rauber, Alexey Kurakin, Nicolas Papernot, Behar Veliqi, Marcel Salathé, Sharada P Mohanty, and Matthias Bethge. Adversarial vision challenge. *arXiv preprint arXiv:1808.01976*, 2018.

[16] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Ipguard: Protecting intellectual property of deep neural networks via fingerprinting the classification boundary. In *Asia Conference on Computer and Communications Security (ASIA CCS), Virtual Event, Hong Kong, June 7-11*, pages 14–25. ACM, 2021.

[17] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, and Aleksander Madry. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

[18] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[19] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE, 2018.

[20] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. Side channel attacks for architecture extraction of neural networks. *CAAI Transactions on Intelligence Technology*, 6(1):3–16, 2021.

[21] Abhishek Chakraborty, Ankit Mondal, and Ankur Srivastava. Hardware-assisted intellectual property protection of deep learning models. In *57th Design Automation Conference (DAC), San Francisco, CA, USA, July 20-24*, pages 1–6. ACM/IEEE, 2020.

[22] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *2009 IEEE International high level design validation and test workshop*, pages 166–171. IEEE, 2009.

[23] Huili Chen et al. Deepattest: an end-to-end attestation framework for deep neural networks. In *46th International Symposium on Computer Architecture (ISCA), Phoenix, AZ, USA, June 22-26*, pages 487–498. ACM, 2019.

[24] Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. Ead: elastic-net attacks to deep neural networks via adversarial examples. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[25] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[26] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[27] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[28] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[29] Joseph Clements and Yingjie Lao. Backdoor attacks on neural network operations. *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 2018.

[30] Joseph Clements and Yingjie Lao. Hardware trojan design on neural networks. In *International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, May 26-29*, pages 1–5. IEEE, 2019.

[31] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[32] Aijiao Cui, Chip-Hong Chang, Sofiène Tahar, and Amr T. Abdel-Hamid. A robust FSM watermarking scheme for IP protection of sequential circuit design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(5):678–690, 2011.

[33] Yaodong Cui, Ren Chen, Wenbo Chu, Long Chen, Daxin Tian, Ying Li, and Dongpu Cao. Deep learning for image and point cloud fusion in autonomous driving: A review. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[34] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. Deepsigns: An end-to-end watermarking framework for ownership protection of deep neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 485–497, 2019.

[35] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

[36] Michael V DeBole, Brian Taba, Arnon Amir, Filipp Akopyan, Alexander Andreopoulos, William P Risk, Jeff Kusnitz, Carlos Ortega Otero, Tapan K Nayak, Rathinakumar Appuswamy, et al. Truenorth: Accelerating from zero to 64 million neurons in 10 years. *Computer*, 52(5):20–29, 2019.

[37] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009.

[38] Lei Deng, Guanrui Wang, Guoqi Li, Shuangchen Li, Ling Liang, Maohua Zhu, Yujie Wu, Zheyu Yang, Zhe Zou, Jing Pei, et al. Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation. *Journal of Solid-State Circuits*, 55(8):2228–2246, 2020.

[39] Yunbin Deng. Deep learning on mobile devices: a review. In *Mobile Multimedia/Image Processing, Security, and Applications 2019*, volume 10993, page 109930A. International Society for Optics and Photonics, 2019.

[40] Priyanka Dixit and Sanjay Silakari. Deep learning algorithms for cybersecurity applications: A technological and status review. *Computer Science Review*, 39:100317, 2021.

[41] Alexey Dmitrenko et al. DNN model extraction attacks using prediction interfaces. 2018.

[42] Xiaoyi Dong, Dongdong Chen, Jianmin Bao, Chuan Qin, Lu Yuan, Weiming Zhang, Nenghai Yu, and Dong Chen. GreedyFool: Distortion-aware sparse adversarial attack. *Advances in Neural Information Processing Systems*, 33:11226–11236, 2020.

[43] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*. OpenReview.net, 2021.

[44] Nikolaus Dräger, Yonghao Xu, and Pedram Ghamisi. Backdoor attacks for remote sensing data with wavelet transform. *arXiv preprint arXiv:2211.08044*, 2022.

[45] Ranjie Duan, Xingjun Ma, Yisen Wang, James Bailey, A Kai Qin, and Yun Yang. Adversarial camouflage: Hiding physical-world attacks with natural styles. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1000–1008, 2020.

[46] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Bomanet: Boolean masking of an entire neural network. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[47] Rameshwar Dubey et al. Blockchain technology for enhancing swift-trust, collaboration and resilience within a humanitarian supply chain setting. *International Journal of Production Research (IJPR)*, 58(11):3381–3398, 2020.

[48] Gamaleldin F Elsayed, Shreya Shankar, Brian Cheung, Nicolas Papernot, Alex Kurakin, Ian Goodfellow, and Jascha Sohl-Dickstein. Adversarial examples that fool both human and computer vision. *arXiv preprint arXiv:1802.08195*, 2018.

[49] Maik Ender, Samaneh Ghandali, Amir Moradi, and Christof Paar. The first thorough side-channel hardware trojan. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 755–780. Springer, 2017.

[50] Yanbo Fan et al. Sparse adversarial attack via perturbation factorization. In *16th European Conference on Computer Vision (ECCV), Glasgow, UK, August 23-28, Part XXII*, volume 12367 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2020.

[51] Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. A performance-sensitive malware detection system using deep learning on mobile devices. *IEEE Transactions on Information Forensics and Security*, 16:1563–1578, 2020.

[52] Nic Ford, Justin Gilmer, Nicolas Carlini, and Dogus Cubuk. Adversarial examples are a natural consequence of test error in noise. *arXiv preprint arXiv:1901.10513*, 2019.

[53] Arturo Geigel. Neural network trojan. *Journal of Computer Security*, 21(2):191–232, 2013.

[54] Samaneh Ghandali, Georg T Becker, Daniel Holcomb, and Christof Paar. A design methodology for stealthy parametric trojans and its application to bug attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 625–647. Springer, 2016.

[55] Justin Gilmer, Ryan P Adams, Ian Goodfellow, David Andersen, and George E Dahl. Motivating the rules of the game for adversarial example research. *arXiv preprint arXiv:1807.06732*, 2018.

[56] Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian Goodfellow. Adversarial spheres. *arXiv preprint arXiv:1801.02774*, 2018.

[57] Xueluan Gong, Qian Wang, Yanjiao Chen, Wang Yang, and Xinchang Jiang. Model extraction attacks and defenses on cloud-based machine learning models. *IEEE Communications Magazine*, 58(12):83–89, 2020.

[58] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[59] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *International Conference on Learning Representations (ICLR)*, 2015.

[60] Jingjing Gu, Binglin Sun, Xiaojiang Du, Jun Wang, Yi Zhuang, and Ziwang Wang. Consortium blockchain-based malware detection in mobile devices. *IEEE Access*, 6:12118–12128, 2018.

[61] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[62] Jia Guo and Miodrag Potkonjak. Watermarking deep neural networks for embedded systems. In *International Conference on Computer-Aided Design (ICCAD), San Diego, CA, USA, November 05-08*, page 133. ACM, 2018.

[63] Kaiyuan Guo, Song Han, Song Yao, Yu Wang, Yuan Xie, and Huazhong Yang. Software-hardware codesign for efficient neural network acceleration. *IEEE Micro*, 37(2):18–25, 2017.

[64] Trung Ha, Tran Khanh Dang, Tran Tri Dang, Tuan Anh Truong, and Manh Tuan Nguyen. Differential privacy in deep learning: An overview. In *International Conference on Advanced Computing and Applications*, pages 97–102. IEEE, 2019.

[65] Song Han et al. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *International Symposium on Field-Programmable Gate Array (FPGA), Monterey, CA, USA, February 22-24*, pages 75–84. ACM/SIGDA, 2017.

[66] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE, 2016.

[68] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[69] Zecheng He, Tianwei Zhang, and Ruby B Lee. Model inversion attacks against collaborative inference. In *Annual Computer Security Applications Conference*, pages 148–162, 2019.

[70] Zecheng He, Tianwei Zhang, and Ruby B. Lee. Sensitive-sample fingerprinting of deep neural networks. In *Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, June 16-20*, pages 4729–4737. CVF/IEEE, 2019.

[71] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[72] Keman Huang, Michael Siegel, and Stuart Madnick. Systematically understanding the cyber attack business: A survey. *ACM Computing Surveys (CSUR)*, 51(4):1–36, 2018.

[73] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

[74] W Ronny Huang, Jonas Geiping, Liam Fowl, Gavin Taylor, and Tom Goldstein. Metapoison: Practical general-purpose clean-label data poisoning. *Advances in Neural Information Processing Systems*, 33:12080–12091, 2020.

[75] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. *Advances in neural information processing systems*, 32, 2019.

[76] M Islabudeen and MK Kavitha Devi. A smart approach for intrusion detection and prevention system in mobile ad hoc networks against security attacks. *Wireless Personal Communications*, 112(1):193–224, 2020.

[77] Weiwen Jiang, Lei Yang, Sakyasingha Dasgupta, Jingtong Hu, and Yiyu Shi. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4154–4165, 2020.

[78] Yier Jin and Yiorgos Makris. Hardware trojans in wireless cryptographic ics. *IEEE Design & Test of Computers*, 27(1):26–35, 2010.

[79] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, June 24-28*, pages 1–12. ACM, 2017.

[80] Poonam Kadian, Shiafali M. Arora, and Nidhi Arora. Robust digital watermarking techniques for copyright protection of digital data: A survey. *Wireless Personal Communications (WPC)*, 118(4):3225–3249, 2021.

[81] Yannic Kilcher. Pytorch_cifar10. `https://github.com/yk/PyTorch_CIFAR10`, 2020.

[82] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[83] Jernej Kos, Ian Fischer, and Dawn Song. Adversarial examples for generative models. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 36–42. IEEE, 2018.

[84] Thanasis Kotsiopoulos, Panagiotis Sarigiannidis, Dimosthenis Ioannidis, and Dimitrios Tzovaras. Machine learning and deep learning in smart manufacturing: the smart grid paradigm. *Computer Science Review*, 40:100341, 2021.

[85] Farinaz Koushanfar. Hardware metering: A survey. In *Introduction to Hardware Security and Trust*, pages 103–122. Springer, 2012.

[86] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[87] Alexey Kurakin, Ian Goodfellow, Samy Bengio, Yinpeng Dong, Fangzhou Liao, Ming Liang, Tianyu Pang, Jun Zhu, Xiaolin Hu, Cihang Xie, et al. Adversarial attacks and defences competition. In *The NIPS'17 Competition: Building Intelligent Systems*, pages 195–231. Springer, 2018.

[88] Griffin Lacey, Graham W Taylor, and Shawki Areibi. Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283*, 2016.

[89] Jinsu Lee, Sanghoon Kang, Jinmook Lee, Dongjoo Shin, Donghyeon Han, and Hoi-Jun Yoo. The hardware and algorithm co-design for energy-efficient dnn processor on edge/mobile devices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(10):3458–3470, 2020.

[90] Julian Leonhard. *Analog Hardware Security and Trust*. PhD thesis, Sorbonne Université, 2021.

[91] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal*, 14(1):653–656, 2019.

[92] Peng Li and Rui Hou. Int-monitor: a model triggered hardware trojan in deep learning accelerators. *The Journal of Supercomputing*, 79(3):3095–3111, 2023.

[93] Wenshuo Li et al. Hu-fu: Hardware and software collaborative attack framework against neural networks. In *Computer Society Annual Symposium on VLSI (ISVLSI), Hong Kong, China, July 8-11*, pages 482–487. IEEE, 2018.

[94] Yue Li, Hongxia Wang, and Mauro Barni. A survey of deep neural network watermarking techniques. *Neurocomputing*, 461:171–193, 2021.

[95] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 382–395. Springer, 2009.

[96] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. *International Symposium on Very Large Scale Integration (ISVLSI)*, 2018.

[97] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating robust dnn with resistance to bit-flip based adversarial weight attack. *IEEE Transactions on Computers*, 2022.

[98] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[99] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *International Conference on Computer-Aided Design (ICCAD)*, pages 131–138. IEEE, 2017.

[100] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. 2017.

[101] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. *Department of Computer Science Technical Reports*, 2017.

[102] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[103] Yu Liu, Yier Jin, Aria Nosratinia, and Yiorgos Makris. Silicon demonstration of hardware trojan design and detection in wireless cryptographic ics. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(4):1506–1519, 2016.

[104] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural trojans. In *IEEE International Conference on Computer Design (ICCD)*, pages 45–48. IEEE, 2017.

[105] Zihao Liu, Qi Liu, Tao Liu, Nuo Xu, Xue Lin, Yanzhi Wang, and Wujie Wen. Feature distillation: Dnn-oriented jpeg compression against adversarial examples. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 860–868. IEEE, 2019.

[106] Bo Luo, Yannan Liu, Lingxiao Wei, and Qiang Xu. Towards imperceptible and robust adversarial example attacks against neural networks. *32nd Association for the Advancement of Artificial Intelligence*, pages 1652–1659, 2018.

[107] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *6th International Conference on Learning Representations*, 2018.

[108] Susmita Dey Manasi and Sachin S Sapatnekar. Deepopt: Optimized scheduling of cnn workloads for asic-based systolic deep learning accelerators. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 235–241, 2021.

[109] Clara Meister, Ryan Cotterell, and Tim Vieira. Best-first beam search. *Transactions of the Association for Computational Linguistics (TACL)*, 8:795–809, 2020.

[110] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[111] Roberto Fernandez Molanes, Kasun Amarasinghe, Juan Rodriguez-Andina, and Milos Manic. Deep learning and reconfigurable platforms in the internet of things: Challenges and opportunities in algorithms and hardware. *IEEE Industrial Electronics Magazine (IEM)*, 12(2):36–49, 2018.

[112] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[113] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[114] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM workshop on artificial intelligence and security*, pages 27–38, 2017.

[115] Hamad Naeem, Farhan Ullah, Muhammad Rashid Naeem, Shehzad Khalid, Danish Vasan, Sohail Jabbar, and Saqib Saeed. Malware detection in industrial internet of things based on hybrid image visualization and deep learning model. *Ad Hoc Networks*, 105:102154, 2020.

[116] VR Niveditha, TV Ananthan, S Amudha, Dahlia Sam, and S Srinidhi. Detect and classify zero day malware efficiently in big data platform. *International Journal of Advanced Science and Technology*, 29(4s):1947–1954, 2020.

[117] Tolulope A Odetola, Faiq Khalid, and Syed Rafay Hasan. Labani: Layer-based noise injection attack on convolutional neural networks. In *Proceedings of the Great Lakes Symposium on VLSI 2022*, pages 143–146, 2022.

[118] Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep learning vs. traditional computer vision. In *Science and information conference*, pages 128–144. Springer, 2019.

[119] Christof Paar. A design methodology for stealthy parametric trojans and its application to bug attacks. *Cryptographic Hardware and Embedded Systems(CHES)*, page 625, 2016.

[120] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[121] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Asia conference on computer and communications security*, pages 506–519, 2017.

[122] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, pages 372–387, 2016.

[123] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. Towards the science of security and privacy in machine learning. *IEEE European Symposium on Security and Privacy*, 2016.

[124] Nicolas Papernot, Patrick McDaniel, Ananthram Swami, and Richard Harang. Crafting adversarial input sequences for recurrent neural networks. In *2016 IEEE Military Communications Conference*, pages 49–54, 2016.

[125] Leandro Parente, Evandro Taquary, Ana Paula Silva, Carlos Souza, and Laerte Ferreira. Next generation mapping: Combining deep learning, cloud computing, and big remote sensing data. *Remote Sensing*, 11(23):2881, 2019.

[126] Victor Henrique Cabral Pinheiro, Marcelo Carvalho dos Santos, Filipe Santana Moreira do Desterro, Roberto Schirru, and Cláudio Márcio do Nascimento Abreu Pereira. Nuclear power plant accident identification system with "don't know" response capability: Novel deep learning-based approaches. *Annals of Nuclear Energy*, 137:107111, 2020.

[127] Ashok Kumar Pundir, Jadhav Devpriya Jagannath, and L. Ganapathy. Improving supply chain visibility using iot-internet of things. In *9th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, January 7-9*, pages 156–162. IEEE, 2019.

[128] Eric Qin et al. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, February 22-26*, pages 58–70. IEEE, 2020.

[129] Sree Ranjani, Maneesh P. K., Nirmala Devi, and Jayakumar M. Golden chip free ht detection and diagnosis using power signature analysis. *Workshop on Reliability Aware System Design and Test*, 2016.

[130] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[131] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. New design strategy for improving hardware trojan detection and reducing trojan activation time. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 66–73. IEEE, 2009.

[132] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[133] Kaveh Shamsi et al. IP protection and supply chain security through logic obfuscation: A systematic overview. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(6):65:1–65:36, 2019.

[134] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *ieee Access*, 7:7823–7859, 2018.

[135] Mohammed Shayan, Kanad Basu, and Ramesh Karri. Hardware trojans inspired IP watermarks. *IEEE Design & Test (D&T)*, 36(6):72–79, 2019.

[136] Yi Shi, Yalin Sagduyu, and Alexander Grushin. How to steal a machine learning classifier with deep learning. In *International Symposium on Technologies for Homeland Security*, pages 1–5. IEEE, 2017.

[137] Cameron Shinn. tiny-tpu. `https://github.com/cameronshinn/tiny-tpu`, 2019. Accessed: 2022-03-17.

[138] Samuel Henrique Silva and Peyman Najafirad. Opportunities and challenges in deep learning adversarial robustness: A survey. *arXiv preprint arXiv:2007.00753*, 2020.

[139] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *International Conference on Learning Representations*, 2015.

[140] Hyun Min Song, Jiyoung Woo, and Huy Kang Kim. In-vehicle network intrusion detection using deep convolutional neural network. *Vehicular Communications*, 21:100198, 2020.

[141] Lu Sun, Mingtian Tan, and Zhe Zhou. A survey of practical adversarial example attacks. *Cybersecurity*, 1(1):1–9, 2018.

[142] Wencheng Sun, Zhiping Cai, Yangyang Li, Fang Liu, Shengqun Fang, and Guoyan Wang. Security and privacy in the medical internet of things: a review. *Security and Communication Networks*, 2018, 2018.

[143] Vivienne Sze, Yu-Hsin Chen, Joel Emer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2017.

[144] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[145] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture (SLCA). Morgan & Claypool Publishers, 2020.

[146] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *International Conference on Learning Representations (ICLR)*, 2013.

[147] Muhammad Imran Tariq, Nisar Ahmed Memon, Shakeel Ahmed, Shahzadi Tayyaba, Muhammad Tahir Mushtaq, Natash Ali Mian, Muhammad Imran, and Muhammad W Ashraf. A review of deep learning security and privacy defensive techniques. *Mobile Information Systems*, 2020:1–18, 2020.

[148] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers (DTC)*, 27(1):10–25, 2010.

[149] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *International Conference on Learning Representations*, 2017.

[150] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, volume 16, pages 601–618, 2016.

[151] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *Symposium on Security and Privacy*, pages 36–52. IEEE, 2018.

[152] Erwei Wang, James J Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter YK Cheung, and George A Constantinides. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Computing Surveys*, 52(2):1–39, 2019.

[153] Junsong Wang, Qiuwen Lou, Xiaofan Zhang, Chao Zhu, Yonghua Lin, and Deming Chen. Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga. In *2018 28th international conference on field programmable logic and applications (FPL)*, pages 163–1636. IEEE, 2018.

[154] Yizhi Wang, Jun Lin, and Zhongfeng Wang. Fpap: A folded architecture for energy-quality scalable convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(1):288–301, 2018.

[155] Yunjuan Wang, Poorya Mianjy, and Raman Arora. Robust learning for data poisoning attacks. In *International Conference on Machine Learning*, pages 10859–10869. PMLR, 2021.

[156] Raniyah Wazirali, Rami Ahmad, Ahmed Al-Amayreh, Mohammad Al-Madi, and Ala' Khalifeh. Secure watermarking schemes and their approaches in the iot technology: an overview. *Electronics*, 10(14):1744, 2021.

[157] weiaicunzai. pytorch_cifar100. `https://github.com/weiaicunzai/pytorch-cifar100`, 2021.

[158] Emily Wenger, Roma Bhattacharjee, Arjun Nitin Bhagoji, Josephine Passananti, Emilio Andere, Haitao Zheng, and Ben Y Zhao. Natural backdoor datasets. *arXiv preprint arXiv:2206.10673*, 2022.

[159] Ross Wightman. PyTorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019.

[160] Eric Wong, Frank R Schmidt, and J Zico Kolter. Wasserstein adversarial examples via projected sinkhorn iterations. *36th International Conference on Machine Learning*, pages 6808–6817, 2019.

[161] Baoyuan Wu and Bernard Ghanem. ℓ_p-box ADMM: A versatile framework for integer programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 41(7):1695–1708, 2019.

[162] Lingfei Wu, Yu Chen, Heng Ji, and Bang Liu. Deep learning on graphs for natural language processing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2651–2653, 2021.

[163] Xi Xiang, Giulia I Corsi, Christian Anthon, Kunli Qu, Xiaoguang Pan, Xue Liang, Peng Han, Zhanying Dong, Lijun Liu, Jiayan Zhong, et al. Enhancing crispr-cas9 grna efficiency prediction by data integration and deep learning. *Nature communications*, 12(1):1–9, 2021.

[164] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially transformed adversarial examples. *International Conference on Learning Representations (ICLR)*, 2018.

[165] Peng Yang, Yingjie Lao, and Ping Li. Robust watermarking for deep neural networks via bi-level optimization. In *International Conference on Computer Vision (ICCV)*, pages 14841–14850. IEEE/CVF, 2021.

[166] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Hardware security and trust: Logic locking as a design-for-trust solution. In *The IoT Physical Layer*, pages 353–373. Springer, 2019.

[167] Jing Ye, Yu Hu, and Xiaowei Li. Hardware trojan in fpga cnn accelerator. In *2018 IEEE 27th Asian Test Symposium (ATS)*, pages 68–73. IEEE, 2018.

[168] Xuwang Yin, Soheil Kolouri, and Gustavo K Rohde. Gat: Generative adversarial training for adversarial example detection and robust classification. In *International Conference on Learning Representations*, 2019.

[169] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 2019.

[170] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[171] Jialiang Zhang and Jing Li. Improving the performance of opencl-based FPGA accelerator for convolutional neural network. In *International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, CA, USA, February 22-24*, pages 25–34. ACM/SIGDA, 2017.

[172] Jialong Zhang et al. Protecting intellectual property of deep neural networks with watermarking. In *Asia Conference on Computer and Communications Security (AsiaCCS), Incheon, Republic of Korea, June 04-08*, pages 159–172. ACM, 2018.

[173] Xiaofan Zhang et al. Dnnbuilder: an automated tool for building high-performance DNN hardware accelerators for fpgas. In *International Conference on Computer-Aided Design (ICCAD), San Diego, CA, USA, November 05-08*, page 56. ACM, 2018.

[174] Xiaofan Zhang, Cong Hao, Yuhong Li, Yao Chen, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. A bi-directional co-design approach to enable deep learning on iot devices. *arXiv preprint arXiv:1905.08369*, 2019.

[175] Xiaoqin Zhang et al. Top-k feature selection framework using robust 0-1 integer programming. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 32(7):3005–3019, 2021.

[176] Bingyin Zhao and Yingjie Lao. Clpa: Clean-label poisoning availability attacks using generative adversarial nets. In *Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*, 2022.

[177] Peng Zhou et al. Unsupervised feature selection for balanced clustering. *Knowledge-Based Systems (KBS)*, 193:105417, 2020.

[178] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.