The Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI-22)

DeepHardMark: Towards Watermarking Neural Network Hardware

Joseph Clements, Yingjie Lao

Department of Electrical and Computer Engineering, Clemson University
Clemson, South Carolina, 29634
jfcleme@g.clemson.edu, ylao@clemson.edu

Abstract

This paper presents a framework for embedding watermarks into DNN hardware accelerators. Unlike previous works that have looked at protecting the algorithmic intellectual properties of deep learning systems, this work proposes a methodology for defending deep learning hardware. Our methodology embeds modifications into the hardware accelerator's functional blocks that can be revealed with the rightful owner's key DNN and corresponding key sample, verifying the legitimate owner. We propose an ℓ_p-ADMM based algorithm to co-optimize the watermark's hardware overhead and impact on the design's algorithmic functionality. We evaluate the performance of the hardware watermarking scheme on popular image classification models using various accelerator designs. Our results demonstrate that the proposed methodology effectively embeds watermarks while preserving the original functionality of the hardware architecture. Specifically, we can successfully embed watermarks into the deep learning hardware and reliably execute a ResNet ImageNet classifier with an accuracy degradation of only 0.009%.

Introduction

As deep neural networks (DNNs) continue to increase in size and complexity, there are growing incentives to deploy machine learning systems to dedicated hardware platforms (Wang et al. 2020). While general-purpose processors are still widely utilized across the field (Jouppi et al. 2017), FPGA and ASIC solutions can provide superior performance and efficiency needed for critical commercial systems (Molanes et al. 2018). 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 (Leonhard 2021). Therefore, it is critical to provide a means for hardware developers to assure the security of a design relinquished to the horizontal supply chain (Yasin et al. 2019; Shamsi et al. 2019).

Hardware watermarking allows designers to place a signature into their hardware intellectual properties (IPs) that verify rightful ownership (Dubey et al. 2020; Pundir, Jagannath, and Ganapathy 2019). Beyond the security implications, watermarks secure the hardware designer's profit incentives and support the field's creative endeavors. Often, conventional hardware watermarking operates on logic circuits by identifying unused states and embeds the signature functionality in them (Cui et al. 2011; Abdel-Hamid, Tahar, and Aboulhamid 2005). Recent works have explored the efficacy of intentionally injecting backdoors into DNN algorithmic IPs for use as a watermark embedded into DNN weights (Adi et al. 2018; Zhang et al. 2018a; Doan et al. 2021; Doan, Lao, and Li 2021). Several other categories of DNN watermarking methods have also been investigated, which are all at the algorithmic level (Fan, Ng, and Chan 2019; Uchida et al. 2017). To the best of our knowledge, watermarking techniques have not been applied to protect DNN hardware IPs, and prior algorithmic approaches do not translate into hardware modifications.

Motivated by a recent work that develops a hardware watermarking technique based on embedding intentional Trojans into hardware IPs (Shayan, Basu, and Karri 2019) and recent studies for injecting hardware backdoors into DNN through the accelerator (Clements and Lao 2019; Liu et al. 2020; Hu et al. 2021; Li et al. 2018), we present a framework for embedding hardware watermarks into deep learning hardware. The main concept leverages hardware backdoors to embed a signature into the hardware by modifications to its functional blocks that can be identified with the owner's *key DNN* and *key samples*. Note that *hardware watermarking* is fundamentally different from *DNN watermarking* which protects the algorithmic IP. Typically DNN watermarks are embedded into the model (i.e., by updating the weights in memory), but hardware-assisted DNN watermarks are also seen. Our signature only alters the protected hardware and so serves as a strong proof of ownership over that hardware. We optimize the embedding using a hardware-aware ℓ_p-ADMM algorithm that reduces the impact of the watermark's hardware overhead. Our hardware modifications are activated under rare input combinations and produce a minimal impact on the design's functionality. Our contributions are summarized below:

- This paper explores, for the first time, the application of hardware watermarking techniques on DNN accelerators. The work proposes a Trojan-inspired methodology that is able to embed backdoor-based watermarks into hardware rather than the model parameters.

- We develop a novel hardware-aware algorithm for embedding watermarks into a DNN model while constraining alterations based on their hardware mapping.
- Our experimental results demonstrate that our methodology minimizes the embedded watermark's impact from both the hardware and algorithmic perspectives while successfully embedding the hardware watermark.

Related Work

DNN Hardware Acceleration

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 (Zhang et al. 2018b; Qin et al. 2020). 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 (Sze et al. 2020). Premium DNN accelerators integrate high-volume computational arrays with well-orchestrated data flows that can maximize the utilization of hardware resources (Sze et al. 2017). As illustrated in Figure 1, 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 (Han et al. 2017).

Hardware Trojan/Backdoor

Hardware Trojans are malicious hardware modifications injected during development across the supply chain. These Trojans can be used to degrade the performance of a design, steal secured information, or give an adversary backdoor access to the device (Tehranipoor and Koushanfar 2010). Trojans are composed of two major components: a trigger and a payload that define the activation criteria and malicious effect, respectively. Because these modifications are designed with an emphasis on stealthiness, hardware Trojans are very difficult to detect and remove, especially in the deep nanometer realm (Jain, Zhou, and Guin 2021). Recently, methodologies for injecting backdoors into DNN models through their hardware accelerators have been developed (Clements and Lao 2019; Liu et al. 2020; Hu et al. 2021; Li et al. 2018). Simultaneously, an additional work has demonstrated that hardware Trojans can be leveraged by a designer to embedding watermarks into hardware IPs (Shayan, Basu, and Karri 2019).

DNN Watermarking

Watermarking is a technique conventionally deployed as a countermeasure to multimedia IP theft (Kadian, Arora, and Arora 2021). 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 (Zhao and Lao 2022; Li, Wang, and Barni 2021). This induces abnormal outputs for specific inputs that can identify the DNN. But such schemes

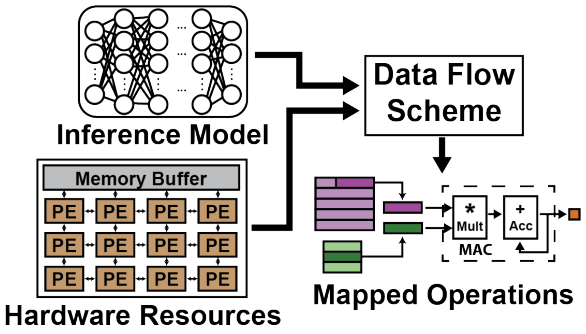

Figure 1: Individual operations in a DNN model will be mapped to specific functional blocks in the hardware based on the available hardware resources and data flow schemes as discussed in (Sze et al. 2017).

are often circumventable by extending the defenses from the adversarial perspective (Adi et al. 2018; Zhang et al. 2018a; Yang, Lao, and Li 2021). DNN fingerprinting (He, Zhang, and Lee 2019; Cao, Jia, and Gong 2021) 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 (Cao, Jia, and Gong 2021). However, these prior works are not applicable for protecting private DNN hardware. Recent works have proposed hardware-assisted DNN obfuscation schemes to protect models (Chakraborty, Mondal, and Srivastava 2020; Chen et al. 2019). These methodology are not targeted at identifying pirated models but degrading performance when used fraudulently.

Problem Setting

Threat Model

In this work, we consider a threat model that is consistent with the literature of hardware watermarking (Shayan, Basu, and Karri 2019). 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 verify the algorithmic IPs and do not extend protection to the hardware. In alignment with prior works (Cui et al. 2011; Shayan, Basu, and Karri 2019), 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, 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.

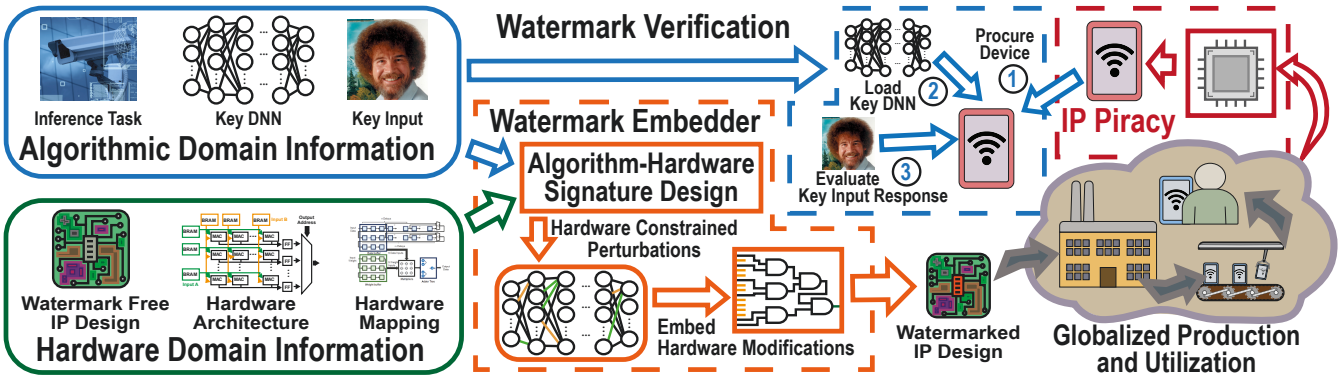

Figure 2: Overview of the proposed algorithm-hardware co-optimized watermarking methodology.

Problem Statement

This paper proposes an algorithm-hardware co-optimized methodology for embedding a hardware watermark into DNN hardware accelerators, as illustrated in Figure 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*, x_k. Ideally, this *signature embedded model*, $F_P(\cdot)$, should only be altered for x_k, described mathematically as:

$$F_P(x) = \begin{cases} y_k, & \text{when } x = x_k, \\ F(x), & \text{otherwise}, \end{cases} \tag{1}$$

where it is required that $y_k \neq F(x_k)$. 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 of in the DNN. These modifications alter the functionality of 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:

$$\boldsymbol{\delta}_k = \boldsymbol{\delta} \odot \mathbf{BH}, \tag{2}$$

where $\odot$ signifies element-wise multiplication.

Equation (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 on the key samples, 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:

$$F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(x) = \begin{cases} y_k, & \text{when } x = x_k, \\ F_k(x), & \text{otherwise}, \end{cases} \tag{3}$$

while any other DNNs executed on the hardware remains unchanged, i.e., $F^{\boldsymbol{\delta} \odot \mathbf{BH}}(x) = F(x)$. As embedding the modifications in the hardware does not require modifying the key DNN or key sample, the execution of $F_k(x_k)$ 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 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, $F_k^{\boldsymbol{\delta} \odot \mathbf{BH}}(x_k) \neq F_k(x_k)$. 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 (Guo and Potkonjak 2018).

Methodology

High-level Overview

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 (Clements and Lao 2019). 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 that 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 $F_k(x_k)$.

Block Constrained Perturbations

The first step in the proposed methodology is to determine a set of perturbations, $\boldsymbol{\delta}_k$, seen in Equation 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:

$$\begin{aligned} \underset{\boldsymbol{\delta},\mathbf{B}}{\text{minimize}} \quad & L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k), \\ \text{subject to} \quad & \mathbf{1}^T\mathbf{B} < c,\ \mathbf{B} \in \{0,1\}^M. \end{aligned} \tag{4}$$

Here L represents a loss function, such as cross entropy loss, that quantifies the watermarking objective with respect to a target output, y_k. $\mathbf{1}^T\mathbf{B} < c$ 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 c 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) 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 (Wu and Ghanem 2019). This method has been broadly employed in many integer programming tasks for its superior performance (Fan et al. 2020; Zhou et al. 2020; Zhang et al. 2021). 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 $\mathcal{S}_p = \{\mathbf{B} : \|\mathbf{B} - \frac{1}{2}(\mathbf{1})\|_2^2 = \frac{M}{4}\}$. A detailed proof of this relationship can be found in the original paper (Wu and Ghanem 2019). 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) to be reformulated as a continuous representation of the MIP problem:

$$\begin{aligned} \underset{\boldsymbol{\delta},\mathbf{B},\mathbf{S}_1\in\mathcal{S}_p,\mathbf{S}_2\in\mathcal{S}_b}{\text{minimize}} \quad & L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(\mathbf{x_k}), \mathbf{y_k}), \\ \text{subject to} \quad & \mathbf{1}^T\mathbf{B} < c,\ \mathbf{B} = \mathbf{S}_1,\ \mathbf{B} = \mathbf{S}_2, \end{aligned} \tag{5}$$

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 1.

Algorithm 1: Block Constrained Perturbations

Require: $F_k(\cdot)$, $L(\cdot)$, $\mathbf{H}$, x_k, y_k
Hyperparameters: c, T_δ, T_B, ϵ_δ, ϵ_B, ρ_1, ρ_2, ρ_3
Ensure: $F_k(x_k) \neq y_k$
$\mathbf{B} = \mathbf{1}$; $\boldsymbol{\delta} = \mathbf{0}$
while $\mathbf{1}^T\mathbf{B} > c$ or $F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k) \neq y_k$ **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 (9)
    Update the dual parameters using Equation (16)
  end for
end while
$\boldsymbol{\delta}_k = \boldsymbol{\delta} \odot \mathbf{BH}$
return $\boldsymbol{\delta}_k$

First, we initialize $\mathbf{B}$ to $\mathbf{1}$ and fix its value. This allows Equation (5) to be simplified to:

$$\underset{\boldsymbol{\delta}}{\text{minimize}} \quad L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k). \tag{6}$$

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 (7):

$$\boldsymbol{\delta} = \boldsymbol{\delta} - \epsilon_\delta \left[\frac{\partial L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k)}{\partial \boldsymbol{\delta}}\right]. \tag{7}$$

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

Second, for a fixed value of $\boldsymbol{\delta}$, Equation (5) simplifies to

$$\begin{aligned} \underset{\mathbf{B},\mathbf{S}_1\in\mathcal{S}_p,\mathbf{S}_2\in\mathcal{S}_b}{\text{minimize}} \quad & L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k), \\ \text{subject to} \quad & \mathbf{1}^T\mathbf{B} < c,\ \mathbf{B} = \mathbf{S}_1,\ \mathbf{B} = \mathbf{S}_2. \end{aligned} \tag{8}$$

This optimization problem should be solved by using the ADMM. The augmented Lagrangian function of Equation (8) can be expressed as:

$$\begin{aligned} &\mathcal{L}(\mathbf{B}, \mathbf{S}_1, \mathbf{S}_2, \mathbf{Z}_1, \mathbf{Z}_2, \mathbf{Z}_3) = L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k) \\ &+ (\mathbf{Z}_1)^T(\mathbf{B} - \mathbf{S}_1) + (\mathbf{Z}_2)^T(\mathbf{B} - \mathbf{S}_2) + \frac{\rho_1}{2}\|\mathbf{B} - \mathbf{S}_1\|_2^2 \\ &+ \frac{\rho_2}{2}\|\mathbf{B} - \mathbf{S}_2\|_2^2 + \frac{\rho_3}{2}(\mathbf{1}^T\mathbf{B} - c) + h_1(\mathbf{S}_1) + h_2(\mathbf{S}_2). \end{aligned} \tag{9}$$

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 (8) is to update $\mathbf{S}_1$ by solving:

$$\mathbf{S}_1 = \underset{\mathbf{S}_1\in\mathcal{S}_p}{\text{argmin}} \ (\mathbf{Z}_1)^T(\mathbf{B} - \mathbf{S}_1) + \frac{\rho_1}{2}\|\mathbf{B} - \mathbf{S}_1\|_2^2. \tag{10}$$

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

$$\mathbf{S}_1 = \mathcal{P}_{\mathcal{S}_p}(\mathbf{B} + \frac{1}{\rho_1}\mathbf{Z}_1). \tag{11}$$

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 (12):

$$\mathbf{S}_2 = \underset{\mathbf{S}_2 \in \mathcal{S}_b}{\operatorname{argmin}} \ (\mathbf{Z}_2)^T(\mathbf{B} - \mathbf{S}_2) + \frac{\rho_2}{2}\|\mathbf{B} - \mathbf{S}_2\|_2^2. \tag{12}$$

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

$$\mathbf{S}_2 = \mathcal{P}_{\mathcal{S}_b}(\mathbf{B} + \frac{1}{\rho_2}\mathbf{Z}_2). \tag{13}$$

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.

$$\mathbf{B} = \mathbf{B} - \epsilon_B \left[\frac{\delta \mathcal{L}}{\delta \mathbf{B}}\right], \tag{14}$$

where

$$\begin{aligned}\frac{\delta \mathcal{L}}{\delta \mathbf{B}} = \frac{\delta L(F_k^{\boldsymbol{\delta}\odot\mathbf{BH}}(x_k), y_k)}{\delta \mathbf{B}} + \rho_1(\mathbf{B} - \mathbf{S}_1) + \mathbf{Z}_1 \\ + \rho_2(\mathbf{B} - \mathbf{S}_2) + \mathbf{Z}_2 + (\rho_3(\mathbf{1}^T\mathbf{B} - c) + \mathbf{Z}_3)\mathbf{1}.\end{aligned} \tag{15}$$

Finally, we update the dual variables with:

$$\begin{aligned}\mathbf{Z}_1 &= \mathbf{Z}_1 + \rho_1(\mathbf{B} - \mathbf{Z}_1) \\ \mathbf{Z}_2 &= \mathbf{Z}_2 + \rho_2(\mathbf{B} - \mathbf{Z}_2) \\ \mathbf{Z}_3 &= \mathbf{Z}_3 + \rho_3(\mathbf{1}^T\mathbf{B} - c),\end{aligned} \tag{16}$$

before recomputing $\mathbf{S}_1$ and $\mathbf{S}_2$ and perturbing $\mathbf{B}$ until a valid solution for Equation (8) is found. We iteratively improve $\boldsymbol{\delta}_k$ by alternating between optimizing Equation (6) and Equation (8) as seen in Algorithm 1.

Intra-block Perturbation Reduction

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:

$$\begin{aligned}\underset{\mathbf{R}}{\text{minimize}} \quad & \|\mathbf{1}^T\mathbf{R}\|, \\ \text{subject to} \quad & F_k^{\mathbf{R}\odot\boldsymbol{\delta}_k}(x_k) = y_k.\end{aligned} \tag{17}$$

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 (Meister, Cotterell, and Vieira 2020).

Algorithm 2: Reducing Intra-block Perturbations

Require: $\boldsymbol{\delta}_k$, $L(\cdot)$, $F(\cdot)$, C
$\mathbf{R}_\rho = \{\mathbf{0}\}$
$\mathbf{R}_N = \{\mathbf{R}_n | \ \|\mathbf{R}_n\|_\infty = 1, \ \mathbf{1}^T\mathbf{R}_n = 1, \ \mathbf{R}_n \odot \boldsymbol{\delta}_k \neq 0\}$
while $F_k^{\mathbf{R}_r\odot\boldsymbol{\delta}_k}(x_k) \neq y_k \ \forall \ \mathbf{R}_r \in \mathbf{R}_\rho$ **do**
  $\mathbf{R}_{\rho_N} = \{\mathbf{R}_r + \mathbf{R}_n | \mathbf{R}_r \odot \mathbf{R}_n = \mathbf{0}, \mathbf{R}_r \in \mathbf{R}_\rho, \mathbf{R}_n \in \mathbf{R}_N\}$
  Loss = [ ]
  for $\mathbf{R}_{rn} \in \mathbf{R}_{\rho_N}$ **do**
    $l_{rn} = L(F_k^{\mathbf{R}_{rn}\odot\boldsymbol{\delta}_k}(x_k), y_k)$
    Loss.append($(\mathbf{R}_{rn}, l_{rn})$)
  end for
  sort_by_loss(**Loss**)
  $\mathbf{R}_\rho = \{$**Loss**$[0 : C-1][0]\}$
end while
$\mathbf{R} = \underset{\mathbf{R}_r \in \mathbf{R}_\rho}{\operatorname{argmin}} \ L(F_k^{\mathbf{R}_r\odot\boldsymbol{\delta}_k}(x_k), y_k)$
return $\mathbf{R}$

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 determine which the choice of $\mathbf{R}^r \in \mathbf{R}^\rho$ and $\mathbf{R}^n \in \mathbf{R}^N$ best minimizes the loss function, $L(F_k^{(\mathbf{R}^r+\mathbf{R}^n)\odot\boldsymbol{\delta}_k}(x_k), y_k)$. 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 2 presents our implementation of this process.

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 (Shayan, Basu, and Karri 2019). 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 that the payload functionality that should be induced in the operation. A trigger and payload can then be designed around this information and embedded in the target functional block to produce the watermarked hardware $H_{\mu_k}(\cdot)$. The specific design depends on the target hardware block and the stealth objectives of the designer. As a case study in this paper, our implementation embeds small combinational logic circuits into the target hardware, as shown in Figure 3. In our example, $\boldsymbol{\mu}_k$ contains observed binary input patterns to an operation when computing, x_k, and bit flip patterns that can produce the perturbation.

Dataset	Model (Acc%)	$\boldsymbol{\rho}_k\% \pm SD$	$ESR\% \pm SD$	ΔAcc% $\pm SD$	ΔFid% $\pm SD$	ΔArea% $\pm SD$
Cifar10	ResNet18 (93)	0.18 ± 0.09	100.0 ± 0.00	0.68 ± 0.14	0.12 ± 0.80	0.22 ± 0.39
Cifar100	ResNet18 (77)	1.29 ± 0.86	100.0 ± 0.00	0.30 ± 0.42	0.25 ± 0.39	1.72 ± 0.72
ImageNet	ResNet18 (89)	0.15 ± 0.07	100.0 ± 0.00	0.67 ± 0.47	0.68 ± 0.47	0.99 ± 0.44

Table 1: Performance of the Proposed Hardware Watermarking on DNN Accelerators

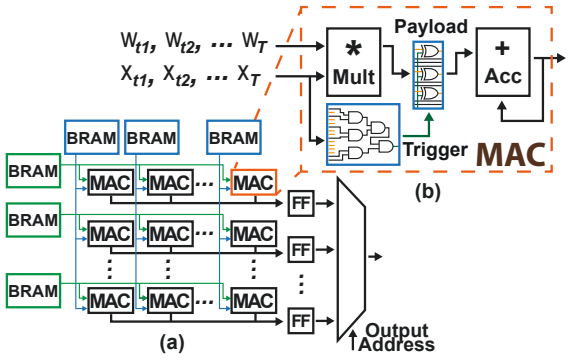

Figure 3: (a) A convolutional neural network hardware accelerator derived from (Zhang and Li 2017). (b) We can embed small combinational circuits into the hardware blocks of the IP. These circuits detect the target input combinations and flip the corresponding output bits as specified by $\boldsymbol{\mu}_k$.

Experimental Evaluation

Experimental Setup

We conduct these experimental evaluations on multiple image classification models for the Cifar10, Cifar100, and ImageNet datasets. The software simulations are developed using the deep learning package, Pytorch. All software simulations are utilized for evaluating the impact of the hardware modifications on the algorithmic functionality of DNN benchmarks consistent with the prior work on hardware-assisted deep learning model obfuscation (Chakraborty, Mondal, and Srivastava 2020). 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.

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 watermarks 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:

$$ESR = \frac{1}{K} \sum_{k=1}^{K} (F_k^{\boldsymbol{\delta}_k}(x_k) == y_k) \times 100\%. \quad (18)$$

  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 on a set of validation data.

$$\Delta\text{Acc}(F_k(\cdot)) = |\text{Acc}(F_k^{\boldsymbol{\delta}_k}(\cdot)) - \text{Acc}(F_k(\cdot))|. \quad (19)$$

  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:

$$\Delta\text{Fid}(F(\cdot)) = |\text{Acc}(F^{\boldsymbol{\delta}_k}(\cdot)) - \text{Acc}(F(\cdot))|. \quad (20)$$

  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 quantify how active the modifications embedded in a design are. The triggering ratio is defined as:

$$T_{ratio} = \frac{\text{\# of times triggered}}{\text{\# of evaluations}} \times 100\%. \quad (21)$$

  The more active the hardware modifications are in a circuit, the more likely it is for them to produce abnormal effects like increased power draw. Ideally, T_{ratio} should be as small as possible.

Efficacy

In Table 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 break down of the individual results, along with the models T_{ratio}, for Cifar10 are shown in Table 3. The value of $\boldsymbol{\rho}_k\%$ represents the percentage of operations in the key DNN that are 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

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 2: FPGA Hardware Overhead. Utilization is reported inside the parenthesis.

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 3: Impact on the Functional Fidelity.

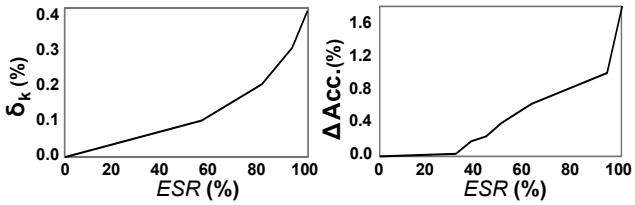

Figure 4: Algorithmic and Hardware Trade-offs

the ESR of the proposed scheme is 100% for all the scenarios evaluated. This is possible because we can relax the carnality constraint, c, in Equation (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.

Trade-offs

In the previous experiments, we ensured a 100% ESR by relaxing the limitation on the cardinality constraint, c. Now we study the relationship between ESR and the methodology's impact on the target hardware under smaller values of c. 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. 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 to 80%.

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 Unit(MMU) similar to (Chakraborty, Mondal, and Srivastava 2020). 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 2. 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 remain the same. The power overhead is also only 0.17%, which further verify the transparency of the proposed hardware watermarking method. In addition, we present the results from ASIC implementation in Table 4, which is based on TinyTPU (Shinn 2019), 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: ASIC Hardware Overhead: TinyTPU.

Conclusion

In this paper, we proposed an algorithm-hardware co-optimized watermarking methodology for DNN accelerators. Based on the mapping from DNN operations to hardware, the algorithm can generate the perturbations that are both limited in the number of hardware blocks that need to be modified and the degree of modifications within each block, allowing for minimal overhead costs when embedding watermarks. Our experimental results have demonstrated the efficacy of the proposed scheme and the preservation of the intended functionality, and the minimal effect of the embedded modifications on the design.

Acknowledgements

This work is partially supported by the National Science Foundation award 2047384.

References

Abdel-Hamid, A. T.; Tahar, S.; and Aboulhamid, E. M. 2005. A Public-Key Watermarking Technique for IP Designs. In *Design, Automation and Test in Europe Conference and Exposition (DATE), 7-11 March, Munich, Germany*, 330–335. IEEE.

Adi, Y.; et al. 2018. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In *27th USENIX Security Symposium, Baltimore, MD, USA, August 15-17*, 1615–1631. USENIX Association.

Cao, X.; Jia, J.; and Gong, N. Z. 2021. 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*, 14–25. ACM.

Chakraborty, A.; Mondal, A.; and Srivastava, A. 2020. Hardware-Assisted Intellectual Property Protection of Deep Learning Models. In *57th Design Automation Conference (DAC), San Francisco, CA, USA, July 20-24*, 1–6. ACM/IEEE.

Chen, H.; et al. 2019. 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*, 487–498. ACM.

Clements, J.; and Lao, Y. 2019. Hardware Trojan Design on Neural Networks. In *International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, May 26-29*, 1–5. IEEE.

Cui, A.; Chang, C.; Tahar, S.; and Abdel-Hamid, A. T. 2011. 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.

Doan, K.; Lao, Y.; and Li, P. 2021. Backdoor Attack with Imperceptible Input and Latent Modification. In *Neural Information Processing Systems (NeurIPS)*.

Doan, K.; Lao, Y.; Zhao, W.; and Li, P. 2021. LIRA: Learnable, Imperceptible and Robust Backdoor Attacks. In *International Conference on Computer Vision (ICCV)*, 11966–11976. IEEE/CVF.

Dubey, R.; et al. 2020. 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.

Fan, L.; Ng, K.; and Chan, C. S. 2019. Rethinking Deep Neural Network Ownership Verification: Embedding Passports to Defeat Ambiguity Attacks. In *32nd Annual Conference on Neural Information Processing Systems (NeurIPS), December 8-14, Vancouver, BC, Canada*, 4716–4725.

Fan, Y.; et al. 2020. 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*, 35–50. Springer.

Guo, J.; and Potkonjak, M. 2018. Watermarking deep neural networks for embedded systems. In *International Conference on Computer-Aided Design (ICCAD), San Diego, CA, USA, November 05-08*, 133. ACM.

Han, S.; et al. 2017. 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*, 75–84. ACM/SIGDA.

He, Z.; Zhang, T.; and Lee, R. B. 2019. Sensitive-Sample Fingerprinting of Deep Neural Networks. In *Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, June 16-20*, 4729–4737. CVF/IEEE.

Hu, X.; et al. 2021. Practical Attacks on Deep Neural Networks by Memory Trojaning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 40(6): 1230–1243.

Jain, A.; Zhou, Z.; and Guin, U. 2021. Survey of Recent Developments for Hardware Trojan Detection. In *International Symposium on Circuits and Systems (ISCAS), Daegu, South Korea, May 22-28*, 1–5. IEEE.

Jouppi, N. P.; et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, June 24-28*, 1–12. ACM.

Kadian, P.; Arora, S. M.; and Arora, N. 2021. Robust Digital Watermarking Techniques for Copyright Protection of Digital Data: A Survey. *Wireless Personal Communications (WPC)*, 118(4): 3225–3249.

Leonhard, J. 2021. *Analog Hardware Security and Trust*. Ph.D. thesis, Sorbonne Université.

Li, W.; et al. 2018. 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*, 482–487. IEEE.

Li, Y.; Wang, H.; and Barni, M. 2021. A survey of Deep Neural Network watermarking techniques. *Neurocomputing*, 461: 171–193.

Liu, Z.; et al. 2020. Sequence Triggered Hardware Trojan in Neural Network Accelerator. In *38th VLSI Test Symposium (VTS), San Diego, CA, USA, April 5-8*, 1–6. IEEE.

Meister, C.; Cotterell, R.; and Vieira, T. 2020. Best-First Beam Search. *Transactions of the Association for Computational Linguistics (TACL)*, 8: 795–809.

Molanes, R. F.; Amarasinghe, K.; Rodriguez-Andina, J.; and Manic, M. 2018. 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.

Pundir, A. K.; Jagannath, J. D.; and Ganapathy, L. 2019. 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*, 156–162. IEEE.

Qin, E.; et al. 2020. 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*, 58–70. IEEE.

Shamsi, K.; et al. 2019. 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.

Shayan, M.; Basu, K.; and Karri, R. 2019. Hardware Trojans Inspired IP Watermarks. *IEEE Design & Test (D&T)*, 36(6): 72–79.

Shinn, C. 2019. tiny-tpu. https://github.com/cameronshinn/tiny-tpu. Accessed: 2022-03-17.

Sze, V.; Chen, Y.; Yang, T.; and Emer, J. S. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12): 2295–2329.

Sze, V.; Chen, Y.; Yang, T.; and Emer, J. S. 2020. *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture (SLCA). Morgan & Claypool Publishers.

Tehranipoor, M.; and Koushanfar, F. 2010. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Design & Test of Computers (DTC)*, 27(1): 10–25.

Uchida, Y.; Nagai, Y.; Sakazawa, S.; and Satoh, S. 2017. Embedding Watermarks into Deep Neural Networks. In *International Conference on Multimedia Retrieval (ICMR), Bucharest, Romania, June 6-9*, 269–277. ACM.

Wang, X.; et al. 2020. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials*, 22(2): 869–904.

Wu, B.; and Ghanem, B. 2019. ℓ_p-Box ADMM: A Versatile Framework for Integer Programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 41(7): 1695–1708.

Yang, P.; Lao, Y.; and Li, P. 2021. Robust watermarking for deep neural networks via bi-level optimization. In *International Conference on Computer Vision (ICCV)*, 14841–14850. IEEE/CVF.

Yasin, M.; Mazumdar, B.; Rajendran, J.; and Sinanoglu, O. 2019. Hardware security and trust: Logic locking as a design-for-trust solution. In *The IoT Physical Layer*, 353–373. Springer.

Zhang, J.; and Li, J. 2017. 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*, 25–34. ACM/SIGDA.

Zhang, J.; et al. 2018a. 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*, 159–172. ACM.

Zhang, X.; et al. 2018b. 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*, 56. ACM.

Zhang, X.; et al. 2021. Top-k Feature Selection Framework Using Robust 0-1 Integer Programming. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 32(7): 3005–3019.

Zhao, B.; and Lao, Y. 2022. CLPA: Clean-Label Poisoning Availability Attacks Using Generative Adversarial Nets. In *Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*.

Zhou, P.; et al. 2020. Unsupervised feature selection for balanced clustering. *Knowledge-Based Systems (KBS)*, 193: 105417.