Secure and Private Machine Learning in Hardware

by

Karthik Ganesan

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy,
Graduate Department of Electrical and Computer Engineering, in the University of Toronto

ABSTRACT

SECURE AND PRIVATE MACHINE LEARNING IN HARDWARE

Karthik Ganesan

Doctor of Philosophy

The Graduate department of Electrical and Computer Engineering

University of Toronto

2024

Machine learning (ML) has seen a tremendous rise in interest in recent years, with neural networks in particular gaining widespread adoption. ML models run on many types of hardware, ranging from low-power IoT devices to powerful GPUs and a plethora of dedicated accelerators. However, their popularity has made systems using ML susceptible to several attacks. To safely enable their widespread use, this thesis details mechanisms to improve the security and safeguard the privacy of ML systems. To this end, we make three contributions: FARO, AESIR and EMPATIC.

Our first contribution (FARO) secures the weights of ML models running on low-power IoT devices against side-channel attacks. Such attacks allow for the entire model to be stolen merely by analyzing several power traces of the device. For the attack to work however, operations must be performed in the same order in each trace. To prevent this, FARO adds hardware to randomly shuffle the order of operations during each run as a functional unit within the CPU. FARO secures the crucial weights of the model, while adding $< 5\%$ area, latency and power overheads.

The second contribution (AESIR) enables edge ML accelerators to run security-critical ML algorithms such as differentially private ML (DP-ML). Current ML accelerator designs lack CPUs and are unable to provide the random noise required by such algorithms. AESIR details simple hardware modifications to enable noise addition from arbitrary distributions. Compared to dedicated on-chip hardware for generating noise, AESIR adds $23\times$ lower area and $40\times$ lower

energy while also being secure against side-channel attacks.

The final contribution (EMPATIC) accelerates adversarial training, used to train models which are more robust against adversarial attacks. We do so by efficiently using the support for high-throughput 8-bit floating point ($FP8$) provided by modern GPUs. Directly training models using $FP8$ is challenging due to the extremely narrow range of this datatype. We propose EMPATIC, where we cluster the training data so that groups of similar inputs are run contiguously. We then calculate the difference within each group – which have a smaller range than the original data – and use $FP8$ for performing computations with these differences. EMPATIC matches the accuracy of models trained using higher precision, while effectively using the higher throughput provided by modern GPUs for $FP8$ computation. Together, our three contributions help improve the security of ML models and protect the privacy of user data used to train ML models.

ACKNOWLEDGEMENTS

First and foremost, I would like to begin by thanking my parents, without whose constant love and support, my academic journey would never have been possible. Next, I want to thank my supervisor, Professor Natalie Enright Jerger for all her support throughout my graduate school experience. I have learned a tremendous amount during my 8 years in graduate school, in large part due to the freedom I had to explore research topics which appealed to me. I am sure that the things I have learned (both technical and otherwise) in this group will help me throughout the rest of my career.

I would also like to thank Professor Andreas Moshovos for serving on both my MASc and PhD committees. I have always enjoyed chatting with Professor Moshovos and hearing his thoughts on a wide range of subjects. While I only met Professor Nicholas Papernot during the last few years of graduate school, he has been amazingly supportive and has always taken the time to meet with me to give me his thoughts about my research.

Outside my committee, I would like to thank Professor Deepa Kundur, who has been incredibly helpful and kind during her time as ECE chair. During every interaction, Professor Kundur has been nothing less than amazingly supportive of me, both in academic and extra-curricular endeavours. I have also enjoyed working with Professor Bruno Korst over several years to totally revamp an upper-year course. I will also miss our unscheduled chats about everything and nothing!

I have also been fortunate to work with several wonderful staff members at UofT. I wish to thank Jessica MacInnis first of all. Over the many years I have worked with Jessica, she has not only been extremely helpful in planning and organizing events but also as someone who is extremely knowledgeable about the workings of both the ECE department and UofT in general. In the office of the Provost, Catherine Gagne was a wonderful ally in my work to improve graduate student mental health on campus. And finally, I would like to thank our group admins – Dubravka Burin and Matt Bhaskar – for all their help with many, many small things including room bookings, conference reimbursements and the like. They were always ready to help out with a smile and made every interaction with them enjoyable.

I would also like to thank all my group and office mates –past and present– for all their help and support over the years. There are far too many to list here but I would like to thank each and every one of them for making the rigours of graduate school less vexing. In particular, however, I wish to list people who helped me with specific problems I faced in my research. Many thanks to Nicholas Giamblanco and Charles Lo for their help brainstorming the mathematical formulation in FARO and to Juan Camillo Vega for his help with the mathematics for the EMPATIC. Lastly, I would like to thank those outside UofT who have been a great support to me during graduate school; Albert and as she so oftens like to remind people, my ‘Canadian mom’ Debbie.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1

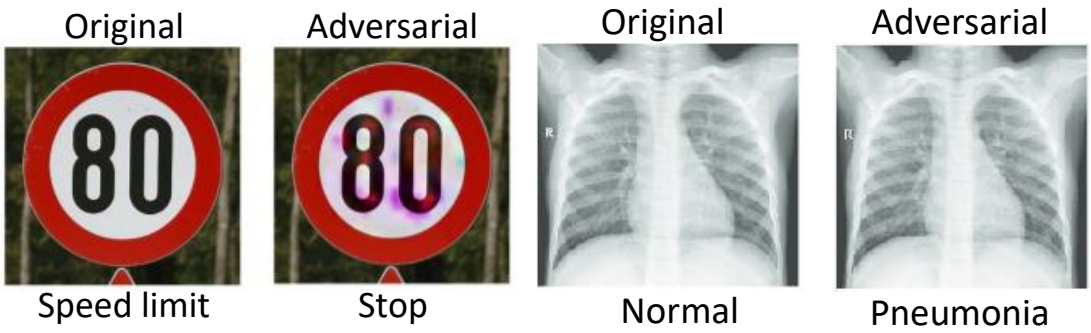
INTRODUCTION

Machine Learning (ML) has emerged in recent years as a transformative technology, advancing the state-of-the-art in diverse domains such as image and voice recognition, indoor localization and biomedical monitoring [187]. ML has also permeated our daily lives in numerous ways. For example, most email programs use ML for identifying spam messages [171, 240]. Voice-to-text – a popular feature on most smartphones – is powered by ML to transcribe speech into text [235]. Such technology underpins popular digital 'assistants' such as Apple Siri and Amazon Alexa [147]. In particular, neural networks have gained significant prominence in recent years and is the focus of this work. While this widespread use of ML has been positive in many ways, it is not without its drawbacks. Ubiquity has made ML the 'common target' for those looking to subvert a wide range of systems.

1.1 Security and privacy of ML

In this work, we consider two complementary aspects of safeguarding ML models; namely the security of the models themselves and the privacy of user data used to train these models. The need to secure ML models is not only of academic interest but has recently become an issue of public policy. For instance, the RAND corporation – a US-based think tank – recently released a report highlighting the pressing need to secure ML models from theft [136]. To understand why it is essential to secure ML models, we ask the question: *why would someone wish to steal a pre-trained ML model?*

We provide two answers to this question. First, training a model to achieve high accuracy takes a significant investment of time and effort. Training often requires one or more GPUs, which can be expensive to purchase or even rent (from cloud service providers). Also, training can be a 'trial-and-error' process, performed multiple times with different training parameters to find the model which achieves the highest accuracy. This iterative process can further increase the cost of training models. Second, model accuracy has been shown to be highly dependent on the quality of the training data [126]. Obtaining high-quality data and processing it to make sure it is suitable for training can incur further costs. For example, a global survey of AI practitioners found that 96% of projects face challenges related to the quality of training data [13]. An attacker who can simply steal a pre-trained model can obtain a highly-accurate model for a fraction of the cost and without requiring high-quality training data.



(a) Altered road signs [66].

(b) Altered X-Ray images [114].

Figure 1.1: Examples of images altered using adversarial attacks. Below each image is how a CNN would classify it.

While some attackers may only be interested in stealing a model to deploy it themselves, for others stealing a model can only be a first step in crafting other attacks. For example, with full access to a trained model, an attacker can learn to craft malicious inputs to fool the model. These malicious inputs can then be used against the model deployed in the field, and cause the model to behave in unpredictable ways. One example is the case of self-driving cars, which use ML to detect and react to road signs. Figure 1.1a shows a speed limit sign (left) and the same sign once it has been subtly altered by an attacker (right) [66]. While a human can still clearly make out the sign on the right as being a speed limit sign, a ML model would classify it as a stop sign. Such a mis-classifications can cause a self-driving car to abruptly speed up or slow down, putting passengers and pedestrians at risk.

Another scenario is insurance companies, who are increasingly using ML to confirm diagnoses made by doctors [70]. Figure 1.1b shows two X-ray images of lungs. The X-ray on the left is the original image, which has been diagnosed as benign by a doctor. However, the X-ray on the right – despite being visually indistinguishable – has been altered to fool a model to classifying it as showing signs of pneumonia. Attackers can alter images submitted for insurance claims to maximize their chances of getting fraudulent payouts, thereby undermining faith in any system powered by ML.

In addition to securing the models themselves, it is also essential that we consider the privacy of user data that was used during training. Particularly for edge applications, sending all user data to a central server for training poses a significant security risk; if the server is hacked, all the user data stored on it is vulnerable. To mitigate this risk, prior works propose ‘collaborative training’, a paradigm where each participant trains a model locally on their own data [169]. The different local models are then shared to generate a final global model, while private data remains with the user. Despite this, collaboratively trained models can still leak user data [174]. For example, Fredrikson et al. show that a neural network trained on user photos can reproduce these photos, given only the user’s name [71]. Thus, we require more robust mechanisms to ensure data privacy when training ML models.

The concerns described in this section highlight the importance of security and privacy for anyone working with ML. The goal of our work is to help improve security and privacy of neural networks specifically, to enable their safe, widespread deployment.

1.2 Contributions

The focus of this thesis is on the security and privacy of hardware used to run ML models. The growing importance of ML has led to a surge of hardware designed and optimized for ML. Power and performance have been the primary design goals for these designs; security and privacy have received far less consideration. We believe that ML security must be viewed holistically – considering the model together with the hardware it is run on – to ensure security and privacy.

Specifically, computer architects working on ML hardware must focus on security and privacy for the following reasons:

1. The choice of hardware used to run the ML model can expose models to new types of attacks. For example, running ML on low-power IoT devices is susceptible to physical side-channel attacks, in contrast to running models on high-power systems such as GPUs (Chapter 3).

2. Dedicated ML hardware may not be able to run ML algorithms designed to protect user privacy (e.g., differentially private ML). Without such support, private user data used in ML accelerators remains vulnerable to theft (Chapter 4).

Furthermore, we believe that knowledge of the underlying hardware can also help speed up techniques which make models safer. We demonstrate this by leveraging hardware support for high-throughput 8-bit floating point in recent GPUs (Chapter 5). With these in mind, we now provide a brief overview of each contribution of this thesis.

1.2.1 FARO

ML models are increasingly used in low-power IoT devices. Prior work shows that ML models deployed on such low-power devices are susceptible to side-channel attacks [18]. By analyzing traces of power consumption while the device is running the ML model, an attacker can reverse engineer all the model weights. However, due to noise from other system components, the attack requires several dozen traces to be analyzed together. For this analysis to succeed, the model weights must be accessed in the same order each time.

Shuffling the order of operations (e.g., the order in which neurons are run in a given layer) prevents the attacker from averaging traces and learning the weights. Prior work performs shuffling in software to prevent this attack [25, 200]. However, we show that software shuffling leads to a tremendous increase in model latency and also leaks side-channel information. This information leakage can then be used to undermine the security offered by shuffling.

To securely and efficiently perform shuffling, we present FARO (Chapter 3). FARO comprises hardware – added as a functional unit within the CPU – to shuffle arbitrary numbers of items (N). Our work is the first to propose hardware to shuffle arbitrary values of N; prior approaches can only perform shuffling is N is a power-of-two. FARO secures the weights of ML models running on IoT devices while adding just 0.56% latency, 2.46% area and 3.28% power overheads.

1.2.2 AESIR

The popularity of ML has driven a surge in accelerators designed specifically for speeding up ML models. However, we find that existing approaches are not designed with security-critical ML

algorithms in mind. Two important algorithms which current edge ML accelerators cannot support are: 1) Differentially private training and 2) Adversarially robust inference. Both algorithms require random values – sampled from specific distributions – for their operation. However, existing techniques for generating such 'noise' in hardware impose significant overheads. In addition, current approaches also leak side-channel information that undermine their security [131, 186].

To enable ML accelerators to support these crucial algorithms, we propose AESIR (Chapter 4). To avoid the drawbacks of generating noise directly in hardware, AESIR pre-computes the required noise ahead of time and stores these values in plentiful off-chip DRAM. Then, when we require noise, we can *randomly sample* values from this stored list. AESIR supports adding noise from arbitrary distributions, while also avoiding the side-channel information leakage that plagues on-chip noise generation. AESIR enables noise addition while incurring 23× lower area and 40× lower energy compared to producing noise directly on-chip.

1.2.3 EMPATIC

Our final contribution focuses on speeding up Adversarial training, a crucial technique for creating ML models which are resilient to adversarial attacks – such as the ones shown in Figure 1.1. Adversarial training improves model robustness by training on inputs which have already been attacked. This requires performing the attacks during training to generate the altered inputs. However, as the attacks are typically iterative, adversarial training incurs a significant performance overhead.

To speed-up adversarial training, we propose EMPATIC, a technique targeting modern GPUs which support high-throughput 8-bit floating-point computations. Prior work has enabled training using 16-bit floating point, but extending this approach to 8-bit is challenging due to the extremely narrow range of 8-bit formats. To overcome this, we first perform cluster the training set to create 'groups' of inputs which are near each other. Within each group, we maintain one input as a 'base' input and calculate the difference between the other inputs and the base input. By doing so, we obtain deltas which (due to our clustering) have a smaller range than the original inputs. We then perform computations using these deltas with using 8-bit values without exceeding the range of this datatype. We show that EMPATIC effectively leverages GPU support for high-throughput 8-bit operations, without suffering the drawbacks of prior approaches.

2

BACKGROUND

In this chapter, we provide some background common to multiple chapters of this thesis. The expert reader may skip the sections they are comfortable with; any background specific to each contribution will be provided at the start of the respective chapter. Here, we start with a brief overview of neural networks – including the most commonly used layers – as well as the process for training neural networks (Chapter 2.1). Finally, in Chapter 2.2, we describe adversarial attacks against neural network as Chapters 4 and 5 focus on techniques to defend against such attacks.

2.1 Neural Networks

Neural networks are a class of machine learning algorithms that see widespread use in modern systems [187]. Neural networks consist of many *neurons* each having several inputs and one output. Figure 2.1a depicts a single neuron, with each operation shown separately for clarity. Each input (x_i) has an associated *weight* (w_i), which determines how strongly that input affects the output of the neuron. We first compute the product of each input with its associated weight and then sum these together. Then, a bias term b is added to this sum, which helps improve the network's training accuracy. To learn non-linear decision boundaries, neurons use an activation function, f. Some common activation functions include the rectified linear unit (ReLU), tanh and sigmoid [187]. In total, the operation of each neuron can be represented mathematically as:

$$y = f\left[\left(\sum x_i \cdot w_i\right) + b\right] \tag{2.1}$$

Neural networks are composed of many 'layers'; each layer is made up of several neurons. Figure 2.1b shows a neural network with three layers: the input layer (which are the inputs to the network in this case), one 'hidden layer' and the output layer (which are the outputs of the network). The middle layer is sometimes referred to as a hidden layer, as it is not visible to the outside like the input and output layers. The strength of neural networks is their ability to learn increasingly complex patterns, as the number of layers and neurons is increased. To understand how, we now provide an overview of how neural networks are trained.

2.1.1 Neural network training

Neural network training is classified under the category of *supervised machine learning*. This means that we require labelled inputs, belonging to the different classes we wish to train with.

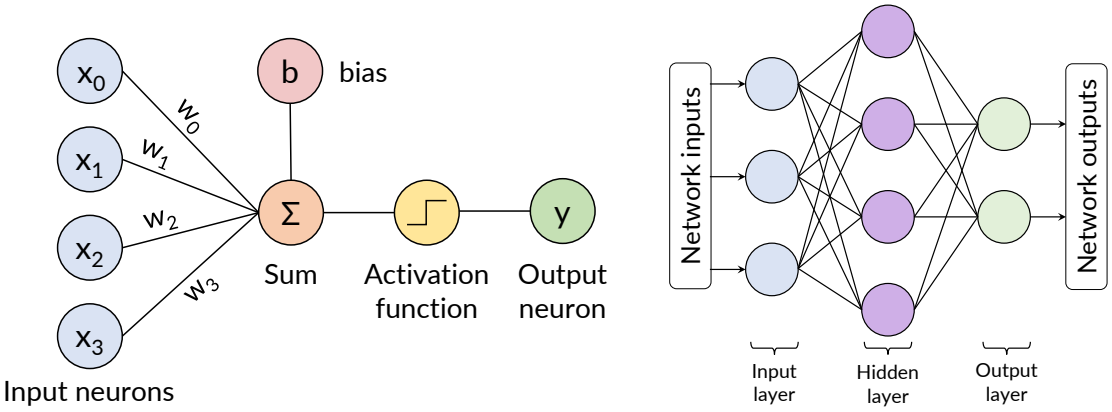


Figure 2.1: (a) One neuron, with four inputs and a bias. (b) A three-layer neural network.

Thus training is performed on a dataset $D = x_1, x_2, ...x_n$ and target labels $L = y_1, y_2, ...y_n$ to find an 'optimal' set of model weights (W), which minimizes the classification error as follows:

$$\underset{W}{\operatorname{argmin}} \sum_{x_i \in D} \mathscr{L}(W(x_i), y_i)) \tag{2.2}$$

The error is calculated using a loss function ($\mathscr{L}$), such as mean squared error or cross-entropy loss [91]. At the start of training, we initialize the model with an starting hypothesis $W_0(x,y)$.

The most commonly used algorithm for training neural networks is Stochastic Gradient Descent (SGD) [91]. We perform a single step of the SGD algorithm to generate a new weight hypothesis $W_1(x,y)$ as:

$$W_2(x,y) = W_1(x,y) - \eta \nabla(W_1(x,y)) \tag{2.3}$$

$\nabla(H_1(x,y))$ is the gradient of the weights with respect to the inputs; this can be thought of as the direction that the weights must change to minimize the loss function $\mathscr{L}$. Thus, for each new training example, the model weights are adjusted to most effectively reduce the loss function. η represents the learning rate of the algorithm which determines how much the hypothesis can change with each training example. In practice however, performing SGD after each training example would be prohibitvely slow. Thus, several training examples are typically grouped into *mini-batches*.

Algorithm 1 shows the computational steps for SGD training, using mini-batches. First, we initialize the layer weights (Line 2) to random values. During each training timestep, we randomly sample minibatches from the full training set (Line 4). For each minibatch, we compute the gradients using the loss function (Line 6). Here, we only show the computation of the gradients with respect to the weights (i.e., ∇_{W_i}). However, as most networks have multiple layers, we must also compute the gradients with respect to the inputs for each layer. This is because the weight gradients computed in layer N are used to update the weights of layer N. On the other hand, the input gradients computed in layer N are used in the backward pass of layer $N-1$. Thus, during back propogation, we compute two sets of gradients in each layer. Once all minibatches are complete, we then aggregate the gradients across all mini-batches (Line 8). Finally, we update

Algorithm 1: Stochastic Gradient Descent (SGD).

1 **Inputs:** Training inputs $(x_1 \dots x_N)$, training labels $(y_1 \dots y_N)$, total timesteps (T), weights (W), minibatch size (B), loss function $(\mathscr{L})$, learning rate (η)
2 **Initialize** model weights before timestep 0: W_0
3 **for** $t \in [T]$ **do**
4 Randomly sample NB minibatches from the training inputs.
5 **for** $b \in [NB]$ **do**
  ▷ Compute gradients
6 $g_t(x_b) \leftarrow \nabla_{W_t} \mathscr{L}(W_t(x_b, y_b))$
7 **end**
  ▷ Sum gradients from all minibatches
8 $\widetilde{g}_t = \frac{1}{B}\left(\sum^{NB} g_t(x_b)\right)$
  ▷ Update weights for next time step
9 $W_{t+1} \leftarrow W_t - \eta \widetilde{g}_t$
10 **end**

the model weights for the next timestep (Line 9).

2.1.2 Common layer types

Neural networks are made up of many *layers*, each consisting of several neurons. The output of neurons from one layer serves as the input to the neurons of the next layer. Below, we list the common types of layers used in neural networks. We show 'C-like' pseudo code for each layer, as we use this code to show our annotations in Chapter 3.

Code Snippet 1: Fully connected layer

```
for i = 0; i < M; i++ do
    for j = 0; j < N; j++ do
        sum[i] += input[j] × weight[i][j];
    end
    sum[i] += bias[i];
    output[i] = actFunc(sum[i]);
end
```

Fully connected layer. The most basic type of layer is the fully connected (or dense) layer. Code snippet 1 shows the code snippet for calculating a single fully connected layer. A dense layer consists of M neurons, with each neuron having N weights. The outer loop iterates over each of these M neurons. In the inner loop, each weight-input pair is multiplied and accumulated to calculate the sum for a single neuron. Finally, a bias value is added to this sum and then passed through the layer's activation function to produce the final output. Networks with just fully connected layers are known as multi-layer perceptron (MLP) networks.

Convolutional layers. Convolutional layers are typically used for classifying spatially or temporally correlated data, such as images, videos or audio. Two dimensional convolutional layers (Conv2D) are the most widely used for image classification. These layers use several 'channels', both for input and output to detect and generate different features. To calculate each output channel, a 2D weight kernel is convolved with each input channel and accumulated. A

Code Snippet 2: 2D Convolutional layer.

```
for oc = 0; oc < M; outch++ do
    for ic = 0; ic < N; inch++ do
        for r = 0; r < R; row++ do
            for c = 0; c < C; col++ do
                for i = 0; i < Kh; i++ do
                    for j = 0; j < Kw; j++ do
                        sum[oc][r][c] += input[ic][S×r+i][S×c+j] × weight[oc][ic][i][j];
                    end
                end
                sum[oc][r][c] += bias[oc];
                output[oc][r][c] = actFunc(sum[oc][r][c]) ;
            end
        end
    end
end
```

bias value is then added and the output is passed through an activation function to get the final output. Code snippet 2 shows the code for a Conv2D layer, implemented using a set of six nested for loops. While these six loops can have any order, in Code snippet 2, we show these loops (starting from outermost) iterating over: output channels, input channels, input height, input width, kernel height and kernel width.

Code Snippet 3: 2D max pooling layer.

```
for ch = 0; ch < N; i++ do
    for r = 0; i < R; r++ do
        for c = 0; j < C; c++ do
            max_value = MAX_NEGATIVE_VALUE();
            for i = 0; i < Kh; i++ do
                for j = 0; j < Kw; j++ do
                    if input[ch][r+i][c+j] > max_value then
                        max_value = (input[ch][r+i][c+j];
                end
            end
            output[ch][r][c] = max_value;
        end
    end
end
```

Pooling layer. To reduce data dimensionality as we move through the network, convolutional layers are typically followed by pooling layers. A pooling layer operates over an $N \times N$ window of input pixels to produce a single final output pixel. Common pooling layers include average and max pooling; these output the average and maximum value in the input window, respectively. Code snippet 3 shows the code for max pooling layer.

Batch normalization layer. As neural networks get deeper, we encounter the problem of vanishing or exploding gradients. This occurs due to the repeated multiply-accumulate (MAC)

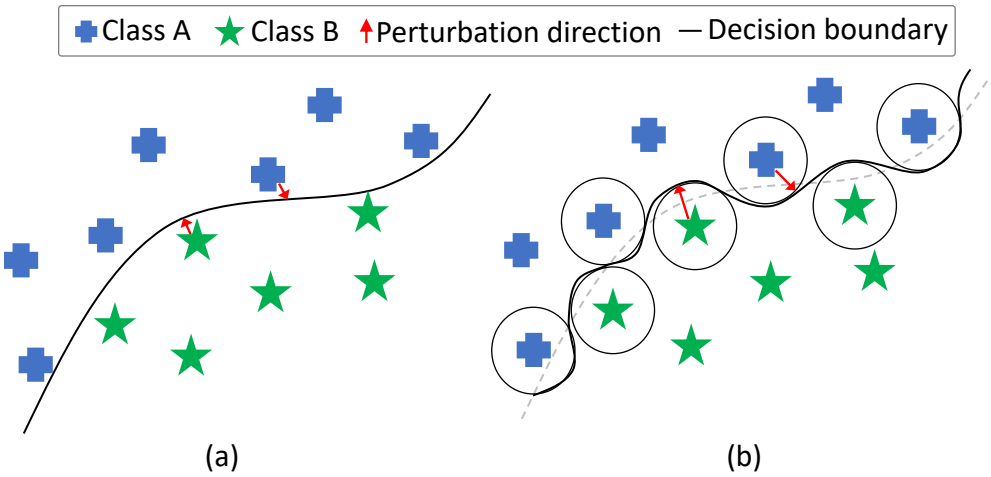


Figure 2.2: (a) Depiction of the decision boundary between two classes and the perturbation needed to cause a mis-classification. (b) Improving model robustness necessitates larger perturbations to nudge inputs across the decision boundary.

operations that occur as inputs run through the network. Over many millions of MAC operations, values above 1 can rapidly increase in magnitude (i.e., 'explode') while values below 1 can correspondingly 'vanish'. A common way to prevent this is to normalize the values after each computational layer using Batch normalization (BatchNorm). During training, BatchNorm computes the mean (μ) and variance (σ^2) of the preceding layer and then normalizes the data to have a mean of 0 and a variance of 1.

During inference, we can normalize values passing through the BatchNorm layer using the trained μ and σ. However, this can add overhead due to the extra operations required. BatchNorm merely performs a simple linear transformation to its inputs (i.e., subtracting the mean (μ) and dividing by the variance (σ)). Therefore, we can avoid the overhead by 'folding' BatchNorm with the preceding layer. This is done by applying the linear transformation directly to the weights of the preceding layer. This is equivalent to running the previous layer and then applying BatchNorm afterwards. We provide a full derivation of how this is done in Appendix A. We adopt this approach in Chapter 3 – which targets inference on low-power IoT devices – and gain the benefits of training with BatchNorm but add no overhead due to this layer during inference.

2.2 Adversarial attacks

In this section, we provide an overview of *adversarial attacks*. We focus on these due to their strength and the difficulty of thwarting such attacks. These attacks predominantly target Convolutional Neural Networks (CNNs) used for image classification [32]. Attackers aim to keep the changes made to the input as small as possible, to avoid detection. For example, in Fig. 1.1a, changing the background colour entirely would be easily spotted so attackers make more subtle changes as shown. We conceptualize this by showing inputs from two classes and the decision boundary separating them in Fig. 2.2(a). The goal of these attacks is to make small changes to 'nudge' the input across this boundary. Thus, for inputs near the boundary, small changes are enough to cause the model to mis-classify, while inputs farther away require larger changes. To understand how attacks make these changes, we first describe the parameter used to

quantify attacks.

Similarity metric: To quantify how much an attack has changed an image, attacks use a 'similarity metric'. The most widely used for this is the distance metric, the L_p norm, given as:

$$|x|_p = (|x_1|^p + |x_1|^p + ... + |x_n|^p)^{\frac{1}{p}} \tag{2.4}$$

Specifically, attacks predominantly use the L_2 and L_∞ norms [32]. For a given image x and an altered image x', the L_2 norm is the Euclidean distance between x and x' while the L_∞ norm is the largest magnitude value in the vector $x - x'$. To quantify the magnitude of these changes, we use a parameter (ε), calculated for the entire image. A higher ε allows for bigger changes, but increases the risk of these changes becoming easier to detect.

Measuring attack success: We now clarify two different accuracy metrics used when discussing adversarial attacks. *Benign* accuracy refers to the accuracy of the model on unperturbed inputs, while *robustness* accuracy refers to the accuracy on inputs which have been attacked. For both accuracies, higher values are better.

Threat model: Finally, we clarify the specific threat model for adversarial attacks in our work. We consider attacks – such as the ones we list below – that have *white-box* access to the model. Under this threat model, the attacker has total knowledge of the underlying model including the architecture, weights and even the algorithm used to train the model (e.g., SGD) [43]. With this unrestricted access, the attacker can query the model as many times as they want. We opt to use white-box attacks are these are the strongest possible types of attacks [43] and are the most widely used in prior work to evaluate the adversarial robustness of networks [65, 109, 128, 149, 160, 249, 263]. We now describe the specific attacks we use in our work.

2.2.1 Common attacks

We now describe the attacks most commonly used in prior work for evaluating robustness.

Fast Gradient Sign Method (FGSM) [92] is a single-step attack that calculates perturbations to a given input. For an input image x of class y, FGSM computes the gradient ∇ with respect to x as:

$$x^{adv} = x + \varepsilon \cdot sign(\nabla_x \mathscr{L}(W(x,y))) \tag{2.5}$$

In Equation 2.5, ε is the attack strength and *sign* is a function that returns just the sign of the computed gradients.

Projected Gradient Descent (PGD) [163] improves on FGSM by iteratively making smaller alterations to the input image. PGD takes several 'steps', where the image is altered by the 'step size' (α) in each step. This contrasts with the FGSM attack which does a single 'step' of size ε. PGD also initializes the attack to a random point within the specified maximum distance (based on the norm).

$$x^{adv}_{st+1} = proj\left\{x^{adv}_{st} + \alpha \cdot sign(\nabla_x \mathscr{L}(W(x^{adv}_{st}, y)))\right\} \tag{2.6}$$

In Equation 2.6, x_{st} is the image at step st. Also, $proj$ indicates that at the end of each attack

step, x^{adv} is clipped so that the perturbations do not exceed ε.

Carlini and Wagner (CW) [31] aims to find the minimum perturbation δ to the image x, such that $x+\delta$ does not exceed the maximum allowable pixel value. CW does so by solving an optimization problem comprising two parts: 1) minimizing the perturbation δ and 2) maximizing the loss for the perturbed input $x+\delta$.

$$min \|\delta\|_p + c \cdot f(x+\delta) \tag{2.7}$$

In Equation 2.7, c is a parameter which controls the weighting of the two parts. Finally, CW uses stochastic gradient descent to solve this optimization problem.

AutoAttack (AA): Despite their popularity, the attacks listed thus far suffer from a few drawbacks. Parameters such as ε, α and c need to be manually chosen, which can lead to sub-optimal attack performance. AA remedies this by proposing a set of 'parameter-free' attacks [46]. Furthermore, instead of consisting of a single attack, AA uses four separate attacks. AA proposes a modified version of the PGD (Equation 2.6), called **APGD**, which automatically tunes the step size during each attack iteration to more efficiently converge on a successful attack image. AA implements two version of APGD using two different loss functions: 1) the commonly used cross-entropy loss and 2) the difference of logits ratio loss (i.e, the difference between the prediction for the true class and the largest non-true-class prediction). AA also uses the **Fast Adaptive Boundary** [45] attack and **Square Attack** [6] to comprehensively evaluate model robustness. In recent years, AA has emerged as the standard attack to evaluate the robustness of ML models.

2.3 Defending against adversarial attacks

Given the strength of adversarial attacks to subvert ML systems, many prior works have proposed techniques to defend against such attacks. We briefly describe some of these techniques here and focus on two in particular – namely adversarial training and noise injection – as these are important for understanding our work in Chapters 3 and 5. Finally, in Chapter 2.3.4, we detail approaches that aim to detect attacked images, instead of directly altering the model itself.

2.3.1 Noise injection for robustness

A widely studied technique for improving model robustness is to add 'noise' (i.e., random values sampled from a specified distribution) to the network [109, 128, 149, 160]. Fig. 2.2(b) pictorially depicts the effect of added noise on the model's decision boundary. Without noise, the model only learns to classify specific points in the input space; this leads to a 'smooth' decision boundary between classes. However, with added noise, the network classifies a small region around each input as belonging to the same class. Thus, small changes which only the move the input within this region would no longer be enough to cause a misclassification. The most commonly used is noise sampled from a Gaussian distribution [42, 109, 128, 149, 160]. As a well-studied and commonly used distribution, the Gaussian distribution allows for mathematical formulations of noise injection for robustness [10, 60, 215, 216].

Algorithm 2: Adversarial training algorithm using the PGD attack.

1 **Inputs:** Examples ($x_1 \ldots x_N$), total timesteps (T), weights (W), minibatch size (B),
2 loss function ($\mathscr{L}$), learning rate (η), attack strength (ε), step size (α), number of steps (NS)
3 **Initialize** model weights before timestep 0: W_0
4 **for** $t \in [T]$ **do**
5 Randomly sample NB minibatches.
6 **for** $b \in [NB]$ **do**
  ▷ `Initialize the attack to a random point within the max distance` ε.
7 $\tilde{x}_b^0 \leftarrow x_b + \mathscr{U}(-\varepsilon, \varepsilon)$
  ▷ `Perform` NS `steps of the PGD attack.`
8 **for** $st \in [NS]$ **do**
9 $\tilde{x}_b^{st+1} \leftarrow proj\left\{\tilde{x}_b^{st} + \alpha \cdot sign(\nabla_x \mathscr{L}(W(\tilde{x}_b^{st}, y)))\right\}$
10 **end**
11 $g_t(\widetilde{x}_b) \leftarrow \nabla_{W_t} \mathscr{L}(W_t(\widetilde{x}_b, y_b))$
12 **end**
  ▷ `Sum gradients from all minibatches`
13 $\widetilde{g}_t = \frac{1}{B}\left(\sum^{NB} g_t(x_b)\right)$
  ▷ `Update weights for next time step`
14 $W_{t+1} \leftarrow W_t - \eta \widetilde{g}_t$
15 **end**

Techniques vary in terms of where they add noise in the network, including at 1) the input [42, 260], 2) after a single layer [65, 149], 3) after every convolutional layer [160] or 4) after every convolutional and fully connected layer [109]. However, techniques that add noise just to the input have been shown to be susceptible to attack [29]. Other techniques add noise to several copies of the input and output the average prediction [42, 160]. However, running multiple copies of the input can add significant latency overhead. Similarly, adding noise after every layer can adversely impact performance, based on the technique used to generate the noise. For example, as we describe in Chapter 4, generating Gaussian noise directly on low-power edge ML accelerators adds significant area, energy and performance penalties. In Chapter 4, we describe hardware to enable noise addition with minimal overheads.

2.3.2 Adversarial training

Another widely used approach for improving model robustness is to directly train the network on attacked images. This technique is known as 'adversarial training' (AdvTrain). The most commonly used attack is the L_∞-PGD attack, which has been shown to be the 'universal' attack for this purpose. Algorithm 2 shows the steps for AdvTrain.

Adversarial training first attacks the inputs in each minibatch before computing the weight gradients. We require additional training parameters for the PGD attack (shown in Chapter 2.2.1), such as: 1) the attack strength (ε), 2) the step size (α) and 3) the number of steps (NS).

The attack itself is broken down into several steps. We first calculate a random starting point for the attack (Line 7). We do this by sampling a random point within a 'sphere' of points within a distance of ε. From this point onwards, we show benign and perturbed images as x and $\tilde{x}$,

respectively. We then perform each step of the PGD attack to iteratively improve $\tilde{x}$ (Line 9). Once the attack is finished, we then compute the weight gradient using the final attacked images (Line 11). The rest of the algorithm is the same as regular SGD.

It is important to note that, although we compute gradients in both Lines 9 and 11, they are not identical. In Line 11, we compute the gradients of the weights with respect to the inputs. We then use these weight gradients to update the model weights. In Line 9, we compute the gradients of the inputs with respect to the weights. During attacks, we only compute the latter as attacks do not alter the model weights directly. However, when training the model itself (i.e., update the model weights), we compute both gradients, similar to Algorithm 1. Adversarial training is the most robust technique proposed so far [190]. Since it was proposed several years ago, no subsequent work has been able to subvert the improvement in robustness given by AT. However, adversarially trained models see a significant reduction in benign classification accuracy.

Improvements to Adversarial training. Many prior works seek to improve the classification accuracy (i.e., accuracy on natural or benign inputs) without sacrificing robustness accuracy (i.e., accuracy on attacked inputs) of adversarial training. Zhang et al. focus on the boundary error, where the distance between the data and the decision boundary is $< \varepsilon$ [263]. They propose TRADES – a modified loss term to reduce boundary error – which achieves 10% higher accuracy compared to regular AT. One problem of TRADES is that the regularization term is designed to push natural examples and their adversarial counterparts together, even if the data is misclassified. Wang et al. propose MART, which prioritizes training on misclassified examples [249].

In Chapter 5, we describe our technique to speed up training, by utilizing the capabilities of the underlying hardware. Crucially, our technique is orthogonal to the techniques discussed above and can be combined with these to further speedup adversarial training times.

2.3.3 Other approaches for robustness

In addition to noise injection and adversarial training, many other techniques have also been proposed to improve model robustness. We now list some of these other approaches, implemented in software and hardware.

2.3.3.1 Software approaches

Software approaches aim to improve model robustness by making algorithmic changes. These techniques are agnostic to the underlying hardware used to execute the model.

Input pre-processing. Soon after adversarial attacks were first proposed, early defenses focused on ways to 'clean' the input to the network. By pre-processing the input before classification, these techniques aimed to reduce the impact of any changes made by the attacker. Such pre-processing techniques include: JPEG compression [61], Fourier transforms [15] and random sampling of the input [97, 258]. Another approach is to use a second ML model to pre-process the input before feeding it to the CNN. Models used for this include other CNNs [232], generative adversarial networks [223], variational auto-encoders [122] and denoising auto-encoders [94, 129, 222]. However, all these techniques add noticable overheads as a significant amount of computation must be done on the input before it can be fed to the network.

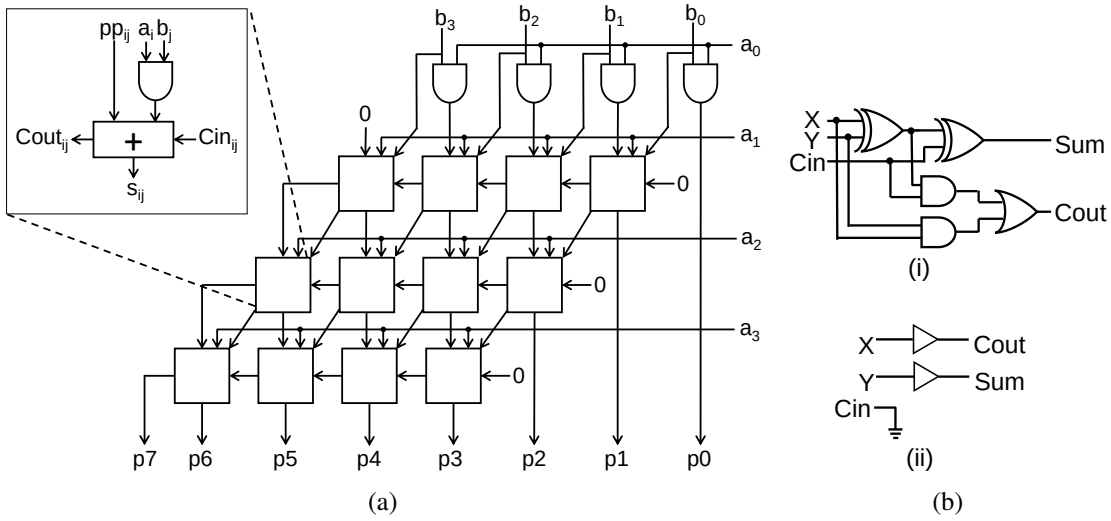


Figure 2.3: (a) A 4×4 array multiplier and (b) A regular full adder (i) and the approximate full adder (ii), used by defensive approximation.

Quantization. Techniques have proposed robustness-aware training for quantized networks [139, 158, 208]. For example, QUANOS trains the model to find the bit-width for each layer that maximizes robustness while maintaining classification accuracy [208]. Defensive Distillation shows that merely quantizing a model actually reduces robustness and adds a new regularization term during training to counter this effect [158]. However, all quantization schemes are static during inference, making them susceptible to attack [20].

Neural architecture search. Another approach for adversarial robustness is to specifically craft network architectures to defend against attacks [33, 35, 98]. Such models have both forward and backward dataflow between layers, compared to typical CNNs which only have forward dataflow. This makes it difficult to efficiently execute these models on most CNN accelerators. NASGuard [246] proposes a new accelerator architecture to more efficiently run such models.

2.3.3.2 Hardware techniques

We now describe some hardware-oriented techniques which have been proposed to defend against adversarial attacks. These techniques are targeted to specific types of hardware (e.g., accelerators) and cannot be applied to other hardware which runs ML.

2-in-1 accelerator (2-in-1) trains models at various bit-widths and randomly selects one model to run each time [74]. Using a random bit-width during inference makes it more difficult for the attack to infer gradients. However, accelerators are not designed to efficiently run models at various bit-widths. Therefore, 2-in-1 also proposes a novel accelerator architecture to support dynamically switching between different bit-widths for each run.

Defensive Approximation (DA) is a hardware technique that uses approximate floating point multipliers for increased robustness [96]. DA targets the 24×24 array-multiplier used to compute the product mantissa in a floating-point multiplier. Figure 2.3a shows a 4×4 array multiplier. DA replaces the full-adders (Figure 2.3b(i)) typically used in an array-multiplier with simple buffers (Figure 2.3b(ii)). This hardware modification adds *input-dependent* randomness into the computation which increases robustness against attacks. This greatly reduces the area and energy

of the floating point hardware.

Using analog noise. Some prior works leverage the noise inherent to performing analog computation to improve robustness. One approach is to under-volt the ML accelerator to reduce the transistor error margins and produce random errors as noise [125, 167]. Another approach is to tune the noise of 6T and 8T SRAM cells, used to store model weights and activations [21]. Roy et al. show that ML accelerators which use crossbars in non-volatile memory to perform calculations can also provide robustness against adversarial attacks [219]. However, achieving robustness using analog noise without sacrificing performance requires careful tuning to account for hardware non-idealities [220].

2.3.4 Detecting adversarial attacks

So far in this section, we have focused on techniques to improve the model's robustness to adversarial attacks. A complementary area of research focuses instead on *detecting* adversarial inputs. Once an input has been identified as malicious, it can be saved for later analysis.

A common approach for identifying malicious inputs is to use a second ML model. Models used for this include: 1) CNNs [179], 2) autoencoders [177], 3) a K-NN classifier [231] or 4) SVMs [162]. However, running a second model alongside the CNN adds considerable overheads. To reduce this overhead, DNNGuard modifies an NVDLA-based accelerator to efficiently run a detection model alongside the regular CNN [247].

Another approach is to use the difference in activation patterns to identify benign vs. adversarial inputs. Ptolemy is a hardware accelerator that stores activations per-layer for each input and then uses a random forest classifier to detect adversarial inputs [77]. DNNShield uses dynamic pruning of the network during inference to identify adversarial inputs [224]. DNNShield performs two inferences runs; one of the baseline network and a second run with dynamic pruning. The difference in classification between both runs is used to identify if the input was adversarial.

Despite being a popular avenue of research, detection techniques must be used with care. Prior works show that detection methods can be bypassed by using a modified loss function for the attack [30]. We therefore focus on techniques which directly improve the robustness of the model.

In this chapter, we presented some background information required to understand the later chapters. We first presented an overview of neural networks, including the training process and commonly used layers. We then described adversarial attacks (which are the focus of Chapters 4 and 5) and prior work on defending against suck attacks. With the background covered, we now move onto our contributions starting with FARO.

3

FARO: HARDWARE SHUFFLING FOR SIDE-CHANNEL SECURITY

The Internet of Things (IoT) has enabled novel applications in fields such as health monitoring [104], smart homes [225] and remote sensing [206]. Within IoT, Machine learning (ML) is seeing increased use in areas such as image and voice recognition, indoor localization and biomedical monitoring [187]. As described in Chapter 1, their widespread use makes ML models a lucrative target of theft. In the case of ML models running on IoT devices, one way to steal the model is to simply read the stored model data from the device memory. However, such direct access to the models stored in on-chip memory is normally blocked by manufacturers. For example, reading memory using the JTAG port on a TI's MSP430FR chips require a password [214]. Thus, attackers must resort to using side channels to steal information from the device.

Side channels are vectors such as timing, power consumption or electromagnetic emanations (EM) which can leak information about data being processed by the device [176]. Prior work shows that side-channel leakage can be used to fully reverse engineer a neural network running on an IoT device [18, 133, 165]. By analyzing EM traces of the device running the network, an attacker can learn the size, activation function, and the weights for every layer. However, for this analysis to succeed, the operations being targeted must occur in the same place in each trace, to remove noise due to other system components. Therefore, one approach to thwart such attacks is to *randomly shuffle* the order of operations each time.

When applied to neural networks, shuffling can lead to a tremendous increase in the number of possible permutations. For example, to reverse engineer M shuffled weights, the attacker would need $\mathcal{O}(M!)$ traces to mount an attack. For a single neuron with 64 weights, shuffling increases the number of traces needed for a successful attack by *90 orders of magnitude*. If an attacker collects and analyzes 1000 traces a second – similar to the speed of our setup – they would need 4.026×10^{78} years to reverse engineer the weights of a single neuron. If we also shuffle the order of neurons (N) per layer, the total number of possible permutations increases to $\mathcal{O}(M!) \times \mathcal{O}(N!)$. As neural networks consist of hundreds of neurons and thousands of weights, collecting enough traces to reverse engineer a whole network would take millions of years, making the attack completely untenable.

Prior work shows that shuffling is effective at preventing side-channel attacks targeting neural networks [25, 200]. However, these works implement shuffling in software, which suffers from a

number of drawbacks:

1. Software shuffling leaks side-channel information. We demonstrate a new attack which undermines the security benefits of software shuffling (Chapter 3.3).
2. Software shuffling adds significant latency overheads due to the additional CPU instructions required (Chapter 3.5).

To overcome the limitations of software shuffling, we propose FARO,[1] hardware to perform random shuffling. FARO is added as a functional unit within the CPU, which significantly reduces the latency overhead of shuffling (Chapter 3.5.3). While prior work has proposed hardware for shuffling, these designs are limited to shuffling 2^N objects [19, 34, 38, 49]. This limitation makes existing approaches unsuitable for shuffling the arbitrary number of weights and neurons used in neural networks. FARO provides an efficient, low-latency hardware solution which supports shuffling any number of values.

Contributions

In this chapter, we make the following contributions:

- We show that shuffling is an effective technique to prevent side-channel attacks against ML algorithms, due to the large number of operations that can be shuffled.
- To the best of our knowledge, we show the first side-channel attack against software shuffling, to learn the exact values being shuffled. An attacker can use this information to 'undo' shuffling and carry out the attack as before.
- To perform shuffling securely and with much less overhead, we add FARO as a functional unit within the CPU. FARO effectively prevents side-channel attacks, while adding just 2.46% area, 3.28% power and 0.56% latency overhead to an ARM M0+ SoC.
- We demonstrate the versatility of our approach by showing that FARO is effective at preventing other side-channel attacks as well as securing other applications against such attacks.

3.1 Background and Related Work

In this section, we provide background on side-channel attacks. We also detail prior work on shuffling, a commonly used technique to defend against these attacks.

3.1.1 Side-channel attacks

Side-channels attacks are a widely used mechanism to obtain secret information about a system, without interfering with normal system operation. Attacks such as SPECTRE [143] and MELTDOWN [159], which target large out-of-order cores, have highlighted the strength of side-channel attacks. SPECTRE and MELTDOWN use *timing* side channels, where an attacker leverages the time difference between certain operations to steal secret information. As IoT systems typically employ very simple processors, they are more commonly targeted by power side-channel attacks [176].

[1] The Faro shuffle – a popular method of shuffling a deck of cards – is conceptually similar to how our hardware shuffles the order in which iterations are run.

Power side-channel attacks require collecting traces of the system being targeted. A trace is a measurement of the device while it is operating on secret data. The power trace varies based on the secret information being operated on by the device. Thus, by analyzing these power traces, an attacker can reverse engineer the secret information used. To collect these traces, an attacker only needs access to the voltage (V_{dd}) input of the device. A commonly used proxy for power is to measure the Electromagnetic (EM) emanations of the device. This does not even require the attacker to physically contact the device at all; the EM probe must simply be placed near the device [176].

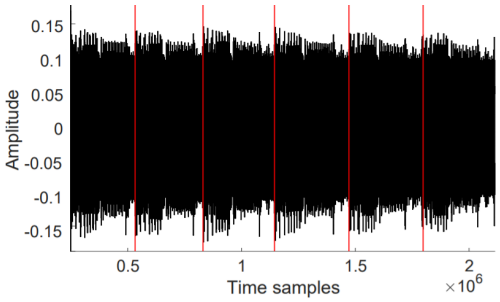
A key difference between timing and power/EM side channels is the number of traces required; with timing channels, information can be leaked with a single trace. However, for power/EM attacks, many traces are required to recover secret information. This is because these side channels are noisy due to interference from other system operations [176]. Thus a single trace does not provide sufficient resolution for an attacker to recover information. The attacker must therefore collect a large number of traces and analyze them together to eliminate noise. Thus, variations between the traces makes the attack more difficult as the attacker must compensate for variations before performing the attack. We now focus on *shuffling* – a popular technique for preventing side-channel attacks – which is the basis of our work.

3.1.2 Shuffling

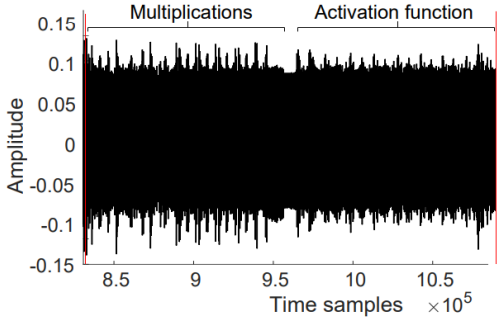
Shuffling randomly reorders the sequence of sensitive operations each time a program is run [170]. With operations happening at different points in each trace, the attacker can no longer identify the position of each operation. Therefore, shuffling N operations forces the attacker to collect $N!$ traces to account for every possible ordering. Shuffling in software was first used to secure the S-Box operation of AES running on a low-power CPU [19]. This is done by unrolling the loop which computes the 16 S-Box computations and running these steps in a random order. However, as we show in Chapter 3.5.3, shuffling in this way adds tremendous overheads when applied to neural networks.

Shuffling neural networks: Prior work has proposed software shuffling for neural networks [25, 200]. However, these approaches suffer from two drawbacks: 1) they add significant latency overheads (as we show in Chapter 3.5.3, this can be as high as 271×) and 2) they leak side-channel information which can undermine their security (Chapter 3.3). This motivates us to design hardware to perform shuffling, to secure ML models against side-channel attacks. Randomly shuffling the order of operations – in software or hardware – requires a random number generator, which we describe next.

True random number generation: To securely produce random numbers, we use a True Random Number Generator (TRNG), a hardware module to produce a sequence of random bits. TRNGs use some physical phenomenon (e.g., power supply noise, temperature, voltage fluctuations) to generate random numbers [234]. By relying on such analog phenomenon, TRNG outputs do not conform to a repeating pattern than an attacker can learn to subvert the security of the TRNG. On supported systems, the TRNG output can be accessed in software using a random number generation function (e.g., *rand*() in C). We assume that both software shuffling and FARO use a TRNG for generating random numbers. Having covered the background and



(a) Identifying the number of neurons in a layer [18]. The red line separates the computation of each neuron.



(b) Identifying the number of weights per neuron and the activation function used per layer [18].

Table 3.1: Comparison of delays (in ms) for commonly used activation functions running on an ARM M3 CPU [18].

Activation function	Min	Max	Mean
ReLU	5.8	6.06	5.9
Sigmoid	152	222	189
Tanh	51	210	184
Softmax	724	877	813

related work for shuffling, we now describe CSI NN, prior work which describes a side-channel attack against neural networks.

3.2 Attacking neural networks

In this section, we explain how the neural network running on an IoT device can be stolen via side-channel attacks. We focus on power/EM side-channel attacks and describe other types of attacks in Chapter 3.6.2. We begin by outlining the threat model we consider in our work (Chapter 3.2.2). While several power/EM side-channel attacks have been proposed [18, 108, 133], we focus on CSI NN [18] as a representative side-channel attack from this class (Chapter 3.2.1). We then describe how we replicate the CSI NN attack (Chapter 3.2.3). Finally, we show how this attack can be extended to ML models other than neural networks (Chapter 3.2.4).

3.2.1 Threat model

To understand the threat model we consider in our work, we must first describe how electromagnetic (EM) side channel attack are carried out. Figure 3.2 shows the steps involved in an EM side channel attack. The IoT system (①) contains the secret information the attacker wishes to steal. We consider a 'passive' attacker, who only observes the device operation but does not interfere with it in any way. Thus, they merely place an near-field EM probe (②) next to the CPU of the IoT system. The data is collected by an oscilliscope (③) and passed to a computer for analysis (④) to obtain the secret information (⑤).

We now detail exactly what information the attacker needs to have to carry out this attack. The attacker knows the program being run on the device. For example, in the case of encryption algorithms, the attacker is aware of when the encryption algorithm is executing on the CPU. In

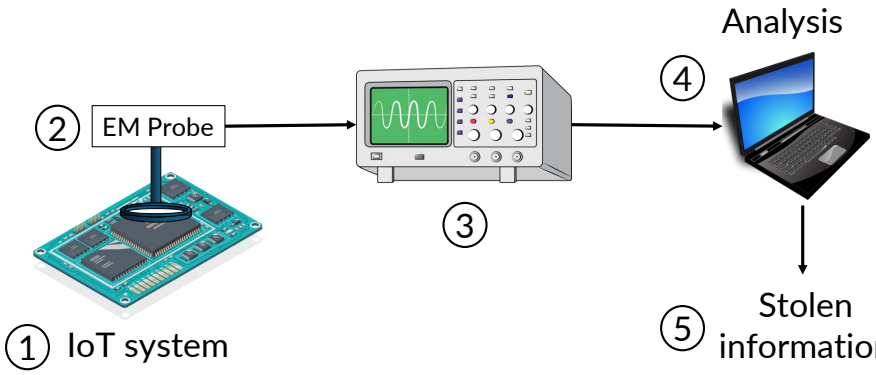


Figure 3.2: Steps to perform an electromagnetic (EM) side channel attack on an IoT device.

this case, the information they wish to steal is the *secret key* used during encryption. They are also able to feed known inputs to the device. This is referred to as a *known plaintext attack* [18]. Finally, we consider a baseline system which has no countermeasures against side-channel attacks. However, we assume that the device will have memory protection to prevent an attacker from directly reading the stored model from non-volatile memory [229].

Specifically for neural networks, our threat model described above is similar to the one described in CSI NN [18] and defences against this attack [25, 200]. As with prior work, we consider MLP and CNN models; we further extend the attack to SVMs and Autoencoders in Chapter 3.2.4. The attacker only knows that the system is running a neural network, but not the specific underlying architecture of the network. They wish to learn the network architecture and the exact weights used in every network layer. We impose no limitations on model size or the activation functions used by networks. Our analysis applies to models using different datatypes, such as floating-point and fixed point numbers.

Sample scenario: The attack scenario we consider arises for IoT devices deployed 'in the field'. For example, consider a network deployed on a wearable device, used for health monitoring. The model may be trained on proprietary training data, which the attacker does not have access to. However, the wearable device itself is readily available, providing the attacker with easy physical access to the device. Thus, an attacker can carry out a side-channel attack against this device by recording and analyzing EM traces of the device to steal the model, without interfering with the normal operation of the device.

3.2.2 CSI NN

CSI NN uses electromagnetic emanations from an IoT device running a neural network to learn the weights and the hyperparameters (i.e., number of layers, number of neurons per layer and the activation functions) of the network. We show how each of these is determined, starting with the network hyperparameters.

Number of neurons. Calculating the output of a neuron consists of several multiplication operations followed by the activation function. Figure 3.1a shows the EM trace for a layer with six neurons. An attacker needs to simply count the number of neurons from the trace.

Activation function. Focusing on the computation of each neuron in Figure 3.1a, we see two distinct regions. Figure 3.1b shows these regions for a single neuron, with the multiplication

operations followed by the activation function. CSI NN observes that common activation functions (i.e., *sigmoid*, *tanh*, *ReLU* and $softmax$) show significant variations in runtime (Table 3.1). *ReLU* takes $<$ 10ms, while *sigmoid* and *tanh* take 50-200ms and *sigmoid* takes 700-900ms. This variation can be used to identify the specific activation function used for each layer. With the activation function known, the attacker can then split the trace into segments containing only the weights for the next step. The next step is to determine the values of these weights using Correlation Power Analysis.

Correlation Power Analysis. Correlation power analysis (CPA) is a type of differential power analysis where the attacker correlates traces using known inputs against a power model to recover the secret data (i.e., weights). CPA consists of 6 key steps, each of which we elaborate on below.

1. Generate a mathematical model of the device's power consumption.
2. Record several traces of the device performing inference using known inputs.
3. Split the traces into segments, with one weight per segment.
4. Attack each weight segment as follows:
   a) Consider every possible option for the weight. For each weight guess and known input, calculate the power consumption according to the model.
   b) Calculate the Pearson correlation coefficient between the modeled and actual power consumption for every data point in the traces.
   c) Pick the weight guess that correlates best to the measured traces.
5. Combine the best weight guesses for all the weights to obtain the full set of weights for the network.

Power model. To carry out a successful CPA attack, the attacker must use an accurate model of the device's power consumption. The power model is highly dependent on the hardware being targeted. In microcontrollers, the memory bus consumes the most power [243]. The memory bus is pre-charged to all 0's before any memory is read. Then, based on the value read, the power consumed is proportional to the number of bus lines that are charged to 1. This is known as the *Hamming Weight (HW)* power model and is the most commonly used model for microcontrollers [18, 165]. The Hamming Weight of a value is given by Equation 3.1.

$$HW(x) = \sum_{i=1}^{n} x_i \tag{3.1}$$

The attacker then generates 'weight candidates' – a list of all possible weight values and their Hamming weights.

Correlating traces. For this step, the attacker first splits each trace into per-weight segments and targets each weight separately. For each weight, the attacker has D power traces (i.e., t), each consisting of T measured data points. The attacker also has a list of I weight values (h), one for each trace (since each trace uses a different input). Now, the attacker must correlate the measured traces t against the guesses of the power model h. The Pearson correlation coefficient (PCC) is the most widely used metric for this purpose [243]. The PCC (ρ) is calculated using

Equation 3.2.

$$\rho_{t,h} = \frac{\sum_d^D [(h_{d,i} - \overline{h_i})(t_{d,j} - \overline{t_j})]}{\sqrt{\sum_d^D (h_{d,i} - \overline{h_i})^2 \sum_d^D (t_{d,j} - \overline{t_j})^2}} \tag{3.2}$$

The parameters in Equation 3.2 are:

- $t_{d,j}$ is point j in trace d.
- $h_{d,i}$ is the weight guess i for trace d.
- $\overline{t_j}$ is the mean of all guesses for each trace d.
- $\overline{h_i}$ is the mean of all guesses for a weight i.

The attacker then uses the absolute value of the PCC to perform the correlation. An absolute value of $\rho_{t,h}$ (i.e., $|\rho_{t,h}|$) close to one means that the weight guess h correlates closely with the trace t, indicating that weight guess is more likely to be the correct guess for that weight. The value with the highest $|\rho|$ is taken as the final guess for that weight. This process is then repeated for every weight in the trace to generate all the weights for the network. In CSI NN, the authors are able to reverse engineer networks with a $< 1\%$ loss in classification accuracy. In contrast, when shuffling is applied using FARO, the recovered weights yield a network with a much lower classification accuracy. For one of the networks we evaluate in Chapter 3.5.3, the accuracy using the recovered weights is just 11.7%.

Number of layers. Figure 3.1a shows a single fully connected layer with six neurons. However, it is not possible to tell this network apart from a network with two layers having three neurons each. In CSI NN, the authors use the $|\rho|$ values to also determine the layer boundaries. The attacker uses a known input to attack all the neurons in the trace. The neurons belonging to the first hidden layer will correlate strongly with the input (i.e., have high $|\rho|$ values). However, as neurons in the second hidden layer do not depend on the input, they show weak correlation. Thus, the last neuron which shows a high correlation marks the layer boundary.

The attacker follows an iterative procedure where they target the first hidden layer, determine its size and recover the weights. Once this is done, they can calculate the outputs of that layer and feed them to the second hidden layer as inputs and repeat the attack. The attacker repeats this process for each layer to reverse engineer the whole network. We now describe how we reproduce the CSI NN attack as a baseline to evaluate FARO.

3.2.3 Reproducing the attack

To reproduce the CSI NN attack, we use the ChipWhisperer CW-NANO platform [205]. The ChipWhisperer is a commonly used platform for side-channel analysis [87, 148, 189]. The CW-NANO platform consists of an ARM M0+ CPU as the 'target' for side channel attacks, alongside an FPGA for data collection and processing.

We collect traces of a MLP network consisting of a 32, 10 and 5 neurons in the input, hidden and output layers, respectively. We first split this trace into segments of just the weights for each neuron and then use correlation power analysis (CPA) to reverse engineer the weights. We empirically determine that 100 traces is sufficient to recover all the weights of the network with 100% accuracy. Our experiments differ from those in CSI NN in two ways: We are able to

recover the weights with 100% accuracy with just 100 traces. In contrast, CSI NN required several hundred traces and weights were not recovered with 100% accuracy. This is due to the following two reasons:

1. CSI NN targeted floating-point values while we target fixed-point operations. Low-power IoT devices typically lack floating-point hardware, which makes fixed-point operations a natural choice for running NNs on these devices.
2. CSI NN used the EM side channel which is more susceptible to noise compared to the power side channel which we use. Despite this, our attack is equivalent to CSI NN since EM is merely a proxy for power.

We also see that the number of traces needed does not scale with the number of weights. This is because each weight is treated independently, thus having more weights does not affect the 'averaging of traces' needed to recover each weight value.

3.2.4 Extending the attack

While CSI NN targets MLPs and CNNs, we show that this attack also works for other ML algorithms, namely autoencoder (AE) networks and support vector machines (SVMs). As AE networks use the same layer types as CNNs, the attack applies directly to them. For SVMs, we target linear kernels (suitable for low power IoT devices), which use two nested for loops. The outer loops iterates over all support vectors and the inner loops over all the input dimensions. The inner loop performs a dot product of the input and a secret weight vector, which an attacker wishes to steal. Thus, shuffling SVMs is similar to shuffling fully connected layers, which are also implemented using two nested for loops. Next, we demonstrate for the first time how shuffling performed in software can still be attacked via side-channel information.

3.3 Attacking Software Shuffling

In this section, we describe how shuffling is implemented in software and how this implementation leaks side channel information. Finally, we outline our attack against software shuffling, which can nullify the security benefits of shuffling in software.

3.3.1 Shuffling for security

Software shuffling has been applied to prevent side channel attacks against neural networks [25, 200]. Both papers shuffle the order of neurons per layer and the order of weights per neuron. Code snippet 4 shows a shuffled implementation of a fully-connected layer with M neurons and N weights per neuron. For clarity, lines added to the baseline code (shown in Code snippet 1) are colored grey. In the un-shuffled case, the next neuron to run is picked by the loop iterator i. With shuffling, we need a separate list to store the shuffled order. Therefore, we make a new list with the values $[0, M)$ in sequence, using the `CreateList` function (Line 1). The new list is then shuffled and for each loop iteration, we read the next element from the shuffled list and run that neuron (Line 4). This process is repeated each time this layer is run, effectively randomizing the order of operations. The weights per neuron are also shuffled in a similar way. Next, we describe how the shuffled list is created in software.

Code Snippet 4: Fully connected layer with software shuffling.

```
1  M_list = CreateList(M)
2  M_shuffled = FisherYatesShuffle(M_list)
3  for i = 0; i < M; i++ do
4      r_i = M_shuffled[i]
5      N_list = CreateList(N)
6      N_shuffled = FisherYatesShuffle(N_list, N)
7      for j = 0; j < N; j++ do
8          r_j = N_shuffled[j]
9          sum[r_i] += input[r_j] × weight[r_i][r_j]
10     end
11     sum[r_i] += bias[r_i]
12     output[r_i] = actFunc(sum[r_i])
13 end
```

Code Snippet 5: Fisher-Yates algorithm for shuffling.

```
1 Function FisherYatesShuffle(list, N):
2     for i = N − 1; i > 0; i−− do
3         j = rand() % (i+1);
4         swap(list[i], list[j]);
5     end
```

Prior work uses the Fisher-Yates algorithm, which is widely used to perform shuffling in security-critical applications such as data and image encryption [3, 106, 221, 236]. Code snippet 5 provides the code to implement Fisher-Yates shuffling for a fully-connected layer.

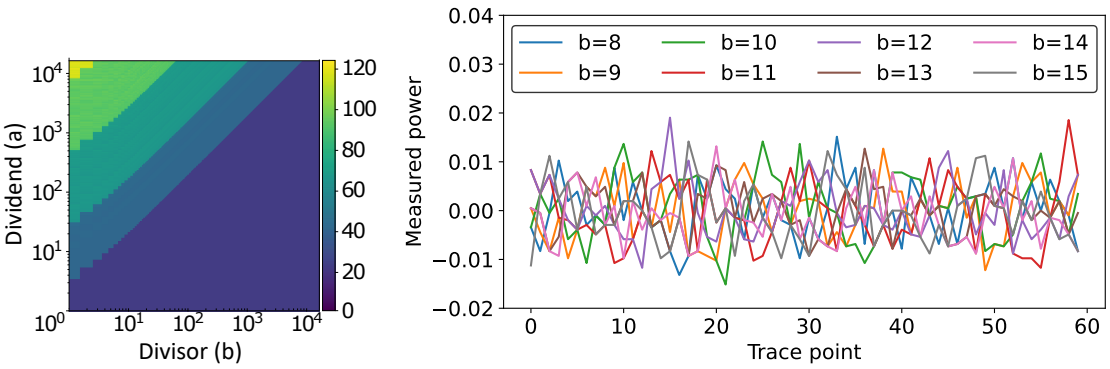
Given a list of N numbers, Code snippet 5 generates a random permutation of this list. Algorithm 5 iterates over every item in the list and for each item, picks a second random item and swaps them. The $rand()$ function queries a l-bit TRNG, which produces a number in the range $[0, 2^l)$ (Line 3). The TRNG output is scaled to the desired range of $[0, i+1)$ with a modulus operation. Finally, the $swap()$ function then swaps both entries (Line 4). When the all iterations are complete, the items in the list indicate the random order in which iterations should be run.

We now focus on the modulus operation, which is the source of the side channel leakage. In hardware, modulus is computed as the remainder of a division operation [52]. Ultra-low power CPUs, such as the ARM M0+ that we use in our evaluation, do not have a hardware divider [52]. They instead implement division in software, using shifts and subtracts.

The ARM GCC compiler for the M0+ CPU uses the `__aeabi_udivmod` function for division and modulus. Code snippet 6 shows pseudo-code for this function. The *division* function computes $a \div b$ and returns the quotient q and remainder (i.e., the modulus) r. The first *While* loop counts the number of steps division will take, by shifting b 1-bit to the left until bit 31 is 1. The number of shifts required is stored in i, which then determines how many times the second *While* loop runs. The second *While* loop performs division by implementing a *restoring division* algorithm. Both the time taken and the power trace vary based on the dividend a and divisor b.

Code Snippet 6: Pseudocode for software division.

```
1  Function division(a, b):
2      if b == 0 then
3          divideByZeroException()
4      else
5          i = 1, q = 0
6          while b[31] ≠ 0 do
7              b = b << 1
8              i = i << 1
9          end
10         while i > 0 do
11             q = q << 1
12             if a ≥ b then
13                 a = a − b
14                 q = q + 1
15             end
16             b = b >> 1
17             i = i >> 1
18         end
19     end
20 return
21 return q, a
```



(a) (b)

Figure 3.3: (a) Heat-map of cycle times for software division, using a log-log scale and (b) Mean subtracted power traces of $100 \div b$ for $b \in [8, 15]$.

3.3.2 Analyzing software division

Latency variation. We begin by profiling the number of cycles taken by software division. We once again use the CW-NANO platform that we described in Chapter 3.2.3. We measure latency using a C program, compiled using the ARM GNU compiler v9.2.1 with *-O3* optimizations. Figure 3.3a shows the heat-map of cycles of $a \div b$ for $a, b \in [1, 16384)$.[2] We see a significant variation in latency when $a > b$ (top left of Figure 3.3a). As Code snippet 6 performs $a - b$ during each iteration, the bigger the value of a compared to b, the more iterations are needed. In

[2]We use 16,384 as that is the maximum number of iterations our implementation supports (Chapter 3.5.3). However, our attack scales to all values of a and b.

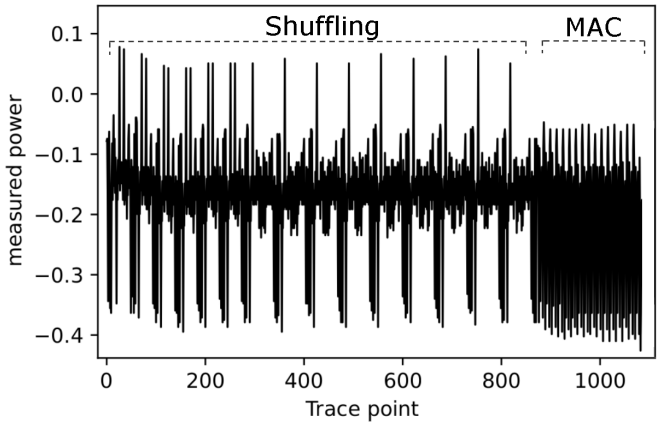


Figure 3.4: Trace showing software shuffling and multiply-accumulate operations for a neuron with 16 weights.

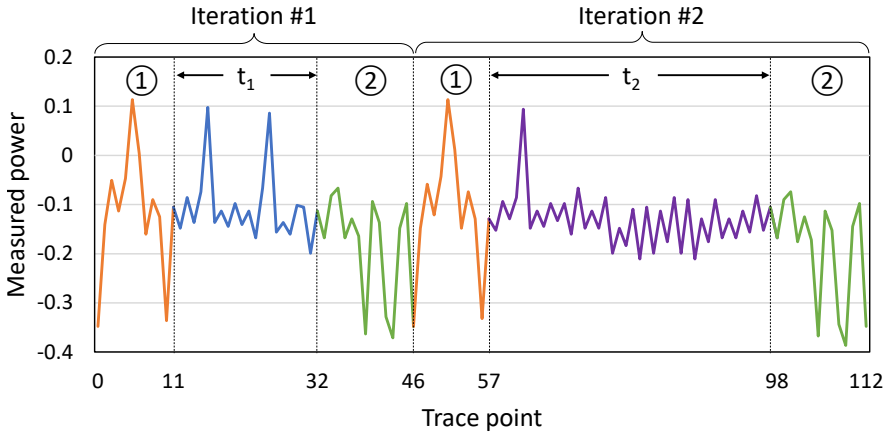


Figure 3.5: Splitting a trace of two loops of software shuffling.

contrast, the latency is similar for all cases where $a < b$ (bottom right of Figure 3.3a). This is because in these cases, Code snippet 6 only runs a single iteration. Since many input values have the same latency, we also analyze the variation in *power* when performing division, to uniquely identify a and b.

Power variation. Figure 3.3b shows mean subtracted power traces for $100 \div b$ for $b \in [8, 15]$. We first take the average of all the power traces (to remove the power contribution of other system components) and then plot each trace minus this average. While dividing 100 by each of these b values has the same latency, we see that the power traces differ based on the value of b, allowing us to tell them apart. While we only show a small range of values for clarity, we see this behaviour for the entire range of inputs we study. Together, we use the input-dependent variation in the latency and power of software division as the basis of our attack.

We could gather and store traces of the system computing $a \div b$ for all possible values of (a, b). Then, we would compare each stored trace against each trace we collect from the system. The stored trace which exactly matches the collected trace would give us the values of a and b. However, this approach suffers from two drawbacks: 1) as a and b can each take 2^N values, the number of comparisons required grows exponentially with N. 2) As Figure 3.3a shows, many values of a and b have the same latency. Thus, the variations in the power traces between these

values is small, making it difficult to uniquely identifying a and b from a single trace this way. We now describe two techniques to narrow the search space and make this identification tractable.

1. Making efficient comparisons. From our earlier profiling of software division, we have a minimum (t_{min}) and maximum (t_{max}) time that division can take. Instead of finding out where division ends, we find where in the collected trace ② begins. As ② is the same for every trace, this comparison is much more efficient. For the first division operation in Figure 3.5, we compute the difference between ② and the collected trace starting at $11+t_{min}$ until $11+t_{max}$. The value of t (i.e., t_{div}) where the difference is 0 gives us the latency of division. Once we know t_{div}, we only need to compare against traces which take that number of cycles. However, as Figure 3.3a shows, many values can have the same division latency. Our second optimization further shrinks the search space.

2. Sequential values. In Code snippet 5, the inputs to the division operation are $rand()$ and $i+1$. We cannot know $rand()$ as it is the output of a TRNG. However, as i goes from N to 1, we know the value of $i+1$ during each iteration.[3] We can learn N by analyzing the traces shown in Figure 3.1b as software shuffling does not obscure the number of items being shuffled. We now know the divisor (b, which is $i+1$) during each division operation. We only need to compare the trace against the stored traces where the divisor is $i+1$, which further reduces the number of comparisons needed.

Training a classifier. With the first two optimizations having reduced the number of comparisons needed, we train decision tree classifiers to predict a, given b and t_{div}. We train our classifiers using SciPy version 1.9.0, using the *gini* criterion. We train a separate classifier for each value of t_{div} and b. By narrowing the range of values that each classifier must predict, we obtain smaller and more accurate classifiers. This allows the classifier to predict the value of a with 100% accuracy.

Note that we get a and b from **a single trace**. CSI NN requires multiple traces because the multiplication operation being targeted is a single-cycle operation. However, as software division takes many cycles, our classifier has many data points it can use, which allows our attack to work with a single trace.

3.3.3 Putting it all together

The target of our attack is the modulus operation (Line 3 in Code snippet 5). We wish to learn the value of j so we know the inputs to the $swap()$ function (Line 4). Using our attack, we find the output of $rand()$ and we also know the value of i. Knowing these values lets us determine the value of j for every iteration. We then collect multiple traces as outlined in Chapter 3.2 and then rearrange each trace based on the swapped indices. With the rearranged traces, we can carry out the power side-channel attack as before. With our attack, software shuffling offers **no security improvement** over the baseline. We therefore conclude from our attack that we require novel hardware which does not leak side channel information for shuffling.

[3] Some implementations of Fisher Yates access items from index 0 to $N-1$. Our approach still applies as elements are accessed sequentially.

3.4 Securing model weights with FARO

In this section, we describe FARO, hardware for efficient shuffling. We begin by outlining the main challenges associated with designing hardware for shuffling. We then provide an overview of our hardware, followed by a description of how software interfaces with our hardware.

3.4.1 Design challenges

Securely shuffling an arbitrary number of iterations in hardware requires overcoming a number of challenges.

Avoiding the memory bus. We want to avoid the memory bus as it is the main source of information leakage. Therefore, we add FARO as a functional unit directly within the CPU. This also reduces the latency of our approach.

Reducing latency. Similar to software shuffling, we could also produce a shuffled list ahead of each loop. However, storing a list of arbitrary size N in hardware is challenging. We cannot store this list in memory as that would require using the memory bus and subject to leaking information. Alternatively, we could use a dedicated on-chip storage but sizing this to accommodate the large dimensions of neural networks would add considerable overhead. As we show in Chapter 3.5.3, our implementation supports $16,384$ iterations for four loops. We use CACTI 7 [17] to determine that storing this many iterations would add 61% area overhead to an ARM M0+ SoC [193]. Instead, we produce random iterations while the layer is running and store the next iteration value in a CPU-accessible register. The CPU reads from this register in a single cycle, thereby minimizing latency and storage overhead.

Avoiding the modulus operation. We must convert the TRNG output from a value in the range $[0,2^l)$ to the range $[0,N)$, using a modulus operation. As we showed in Chapter 3.3, modulus (implemented as division) is susceptible to side-channel attacks. Therefore, we need a way to randomize the order of iterations without using a modulus operation. We now describe FARO, which addresses these challenges without incurring significant overheads.

3.4.2 High level overview

To generate iterations in the range $[0,N)$ in a random order, without repetitions, we first split the total number of iterations into k 'bins'. Each bin then represents a subset of the total number of iterations that must be run. For example, for a single neuron with ten weights, we split them into two bins: bin 0 for iterations 0-4 and bin 1 for iterations 5-9.[4] To start with, each bin is set to its minimum value (i.e., 0 and 5). To pick an iteration to run, we pick one of the two bins and the value in that bin is output. Next, the value in that bin is incremented, ensuring a unique output each time. The process repeats ten times to output ten total iterations, with one of the two bins picked randomly each time. In essence, FARO converts the problem of selecting an iteration in the range $[0,N)$, to picking from a much smaller number of bins. By restricting the number of bins to always be a power of 2, *we can directly use the output of the TRNG without requiring*

[4]The number of bins is configurable at design time. We use a design with 2 bins in this example for clarity.

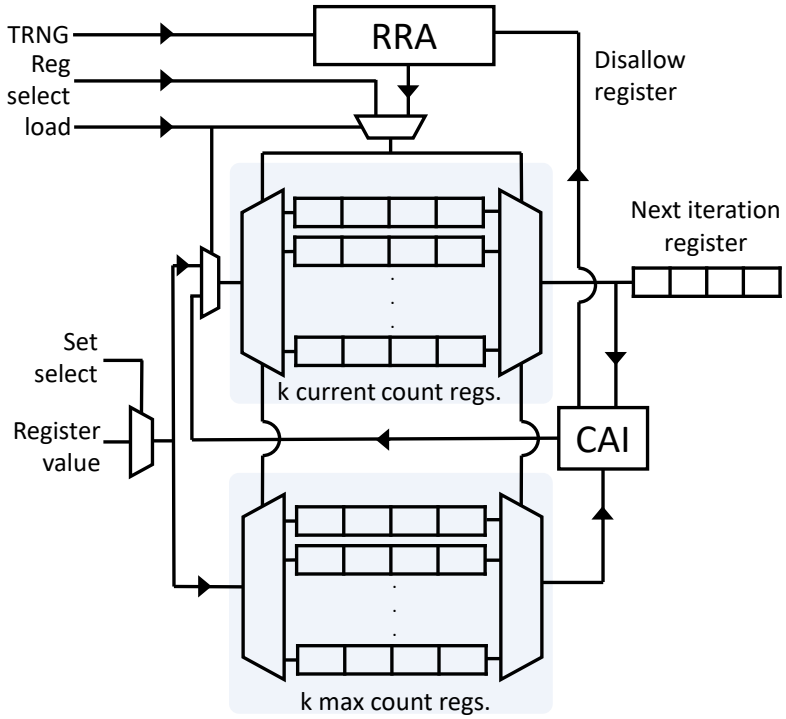


Figure 3.6: Hardware for counter-based shuffling.

a modulus operation. Next, we quantify the total number of possible permutations when using FARO.

Mathematical formulation. When N is a multiple of k, all bins will have the same number of iterations. But when N is not a multiple of k, there will be one bin with fewer iterations. In this case, the first $k-1$ bins will each have a iterations where $a = \lceil N/k \rceil$, while the last bin will have b iterations, where $b = N-(k-1)a$. If N is a multiple of k however, $a = N/k$ and $b = 0$. Therefore, the total number of permutations, P is:

$$P = \begin{cases} N!/[(a!)^k], & \text{if N is a multiple of k} \\ N!/[(a!)^{k-1} \times b!], & \text{otherwise} \end{cases} \tag{3.3}$$

For the case with a single register ($k = 1$), we have $a = N$ (i.e., N iterations all in one bin) and $b = 0$. This gives us just 1 possible order, which is the same as the baseline case. Using N registers per set (i.e., $k = N$), with $a = 1$ (i.e., 1 bin per iteration) and $b = 0$, gives us the maximum possible $N!$ permutations. Using Equation 3.3 for our example above (N=10 and k=2), we get $P = 252$ possible orderings. For larger sizes of N and k, P quickly grows into the millions, which effectively randomizes the sequence. In Chapter 3.5.2, we show how such huge values of P make the attack take intractable lengths of time. We now describe our hardware implementation of this 'bins' to track iterations.

3.4.3 Hardware overview

Figure 3.6 shows an overview of FARO. We use a set of k registers to keep track of the value of each bin. In our example above, we would use two *current count* registers, to track the current

value of each bin. These two current count registers are initially loaded with the values 0 and 5, respectively. We use a TRNG to pick a current count register to output the next iteration. As registers are picked, their values are incremented each time. However, once a current count register reaches its maximum value (i.e., 4 or 9 in our example), we must disallow it from being run again. To keep track of the maximum values for each current count register, we use another set of *max count* registers.

As current count registers begin to saturate, the output from the TRNG will pick disallowed registers. To quickly pick another valid register to run, we employ a combinational Round Robin Arbiter (RRA). The RRA keeps track of all current count registers, using a single bit set to '1' per register, indicating that this current count register still has iterations that can be run. The output of the TRNG is fed to the RRA to pick a current count register. If the corresponding RRA bit is '1', that register's value is output. Next, we compare the value in that current count register with its corresponding max count register. If the maximum value has not been reached, the current count register is incremented. This is performed using the 'compare and increment' (CAI) block in Figure 3.6. However, if a register has reached its maximum value, the CAI block sets the bit corresponding to that register in the RRA to '0', via the 'disallow register' signal. If a disallowed register is later picked, the RRA outputs the closest allowable register to be run instead. The RRA is purely combinational and therefore returns a valid register in a single cycle each time.[5] The number of registers per set is a parameter that can be configured at design time. The larger the number of registers, the more security our design provides but at the cost of increased area. To balance security and added area, we opt for 16 registers per set.

Hardware banks. The hardware shown in Figure 3.6 generates random iterations for a single loop. However, neural network layers are implemented as a series of nested loops. Therefore, we use one copy of the hardware in Figure 3.6 per loop that we wish to randomize. Each loop is associated with one bank and we use multiplexors to pick which bank to use for each loop. We opt for a design which uses four banks, to balance security vs. area and latency overhead. We use two banks for fully connected layers. For convolutional layers, we opt to four out of the six loops. Therefore, we loop over input channels, the output channels, input rows and input columns. Lastly, for max pooling layers, we use three banks to shuffle rows, columns and channels. Shuffling of these loops is achieved by modifications to the code shown using highlighted boxes in Code snippet 7. Next, we describe the purpose of these code additions.

3.4.4 System interface

In this section, we show how FARO is controlled via software and the necessary extensions to support this.

Code annotations. We program FARO via two functions: *load_bank* and *get_next_iteration*. Code snippet 7 shows the code for a fully connected layer with our changes highlighted.[6] The *load_bank* function (lines 1 and 2) loads the registers in a specified bank, before running the loops. The *get_next_iteration* function (lines 4 and 6) queries the hardware for the next iteration from a given bank. The values returned from the *get_next_iteration* are stored in *r_i* and *r_j*

[5] This avoids repeatedly querying the TRNG for a valid register which would be time consuming.
[6] We provide code for 2D convolutional and max pooling layers in Appendix B.

Code Snippet 7: Fully connected layer with FAROfunctions added.

```
1  load_bank(BANK0,M)
2  load_bank(BANK1,N)
3  for i = 0; i < M; i++ do
4      r_i = get_next_iteration(BANK0)
5      for j = 0; j < N; j++ do
6          r_j = get_next_iteration(BANK1)
7          sum[r_i] += input[r_j] * weight[r_i][r_j]
8      end
9      sum[r_i] += bias[r_i]
10     output[r_i] = actFunc(sum[r_i])
11 end
```

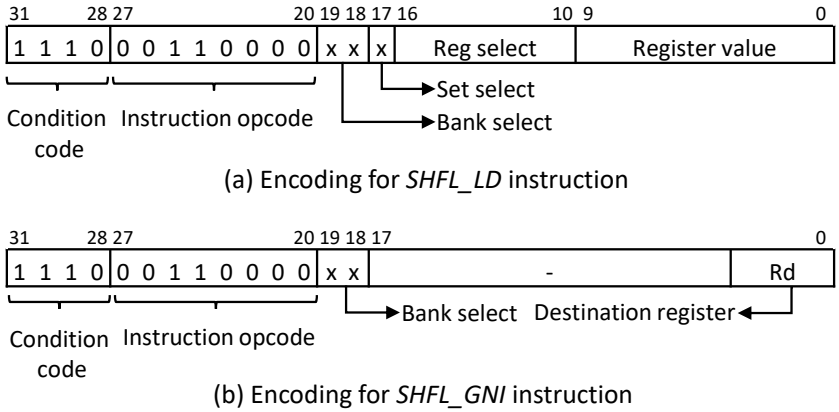


Figure 3.7: Custom ISA instructions for shuffling hardware.

and then used in the loops instead of the original loop iterators i and j. The *load_bank* and *get_next_iteration* functions are defined in a library that we provide. Our library implements these functions using custom ISA instructions, which we describe next.

ISA extensions. We add additional CPU instructions to interface with FARO (Figure 3.7). The first instruction, *SHFL_LD*, loads initial values to the current count and max count registers before each layer. The bits of the *SHFL_LD* instruction are:

- [31:28] represent condition codes that the instruction must check before execution. We set these bits to '1110', as per the ARM Technical Reference Manual [11].
- [27:20] shows an unused opcode in the baseline ARM ISA which we use for our instructions.
- [19:18] select the bank we want to access.
- [17] select the set (i.e., current/max count registers).
- [16:10] specify the register within the set.
- [9:0] are the value to be loaded into the selected register.

The second instruction, *SHFL_GNI*, returns the next iteration from one of the banks. This instruction format is:

- [31:18] are identical to the *SHFL_LD* instruction.
- [17:4] are unused in this instruction.
- [3:0] specifies a CPU register for the result.

The *SHFL_LD* instruction uses 10 bits for the register value, which allows each register to count up to 1024 iterations. We use 7 bits for the register select, which allows for designs with up to 128 registers per set. This instruction encoding therefore supports loops with up to 131,072 iterations. As this is much larger than networks run on an IoT device, this encoding does not limit the size of networks that our technique can support. For example, the largest layer we run in Chapter 3.4.5 is an order of magnitude smaller than the maximum iterations supported by our encoding. Our technique does not impose any restriction on the number of layers nor the total number of weights a network can have.

We provide definitions for the *load_bank* and *get_next_iteration* function calls. The *load_bank* function calculates and loads (using *SHFL_LD*) the current count registers and max count registers. The *load_bank* function is only called once per bank, before each layer. The overhead of *load_bank* scales with the number of registers but is not affected by the size of the layers. Thus, layers of any size require the same number of instructions for the loading operation, which amortizes the overhead of *load_bank*.

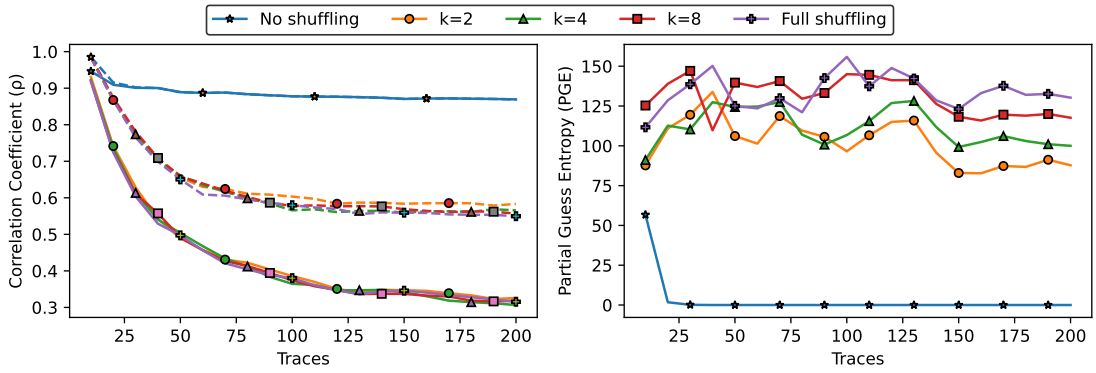
The *get_next_iteration* function performs a single register read and therefore adds just one *SHFL_GNI* instruction to the program binary. However, as the *SHFL_GNI* instruction is called for every single loop iteration, it is critical that we minimize the cycle count of that instruction, to reduce the overall latency impact. Next, we explain how FARO achieves this goal of minimizing the latency of the *SHFL_GNI* instruction.

3.4.5 Hardware latency

We design FARO to provide the next iteration number to the CPU with a one-cycle latency. To do this, we take advantage of the time between subsequent calls to the *get_next_iteration*. In Code snippet 7, we first load BANK0 (line 1). As soon as BANK0 is loaded, the hardware begins selecting the next iteration for that bank. In the meantime, the CPU is loading values for BANK1 (line 2). Thus, we have several cycles to pick the next iteration for BANK0 before it is queried by the CPU.

Similarly, there are seven cycles between subsequent calls to the *get_next_iteration* function, even in the inner for loop (line 5). This is because CPU must do several operations (i.e., calculating the index of the next weight, loading that weight and associated input, performing a multiplication and addition) before requiring the next iteration number. This allows FARO to select the next iteration in time for the next request from the CPU. FARO takes a total of three cycles to generate the next iteration, which gives us several cycles of buffer before the next iteration is queried.

Once calculated, the next iteration value is stored in the 'next iteration' register in each bank until it is read by the CPU. The CPU can then read from this register using the *SHFL_GNI* instruction in a single cycle. As soon as this register is read, the hardware begins selecting the next iteration to run for that bank. This allows us to minimize the latency of the *get_next_iteration* to a single cycle.



(a) Average correlation coefficient (ρ)

(b) Average Partial Guess Entropy (PGE)

Figure 3.8: Effect of different degrees of shuffling for 16 weights on (a) average correlation coefficient (ρ) and (b) average partial guess entropy (PGE).

3.5 Evaluation

In this section, we evaluate the effectiveness of FARO in preventing side-channel attacks. We begin by showing that FARO masks the side channel and secures neural networks against attackers. Next, we show how FARO greatly increases the time needed to collect enough traces to carry out the attack. We then report the runtime overhead of FARO on several representative neural networks and compare against the overhead of software shuffling. Finally, we quantify the area and latency overhead of FARO and explain how FARO does not leak side channel information.

3.5.1 Efficacy of hardware shuffling

We first show how shuffling impacts the effectiveness of the power side-channel attack. Since our target CPU does not have shuffling hardware, we calculate a shuffled order of weight accesses for each run and load this into our CPU prior to trace collection. Thus the traces we collect have the weights accessed in a new shuffled order during each run. For robustness, we collect 200 traces for each run of the network, exceeding the 100 traces we need for the baseline attack. We see that even increasing to 1000 traces does not change our results. Thus, we do not believe that an attacker can circumvent our solution by simply increasing the number of traces they collect.

Effect on ρ. First, we see how shuffling affects the Pearson correlation coefficient (ρ), which is the metric used by CPA to determine the most likely value of each weight. We demonstrate using a single run with 16 weights, where we vary the amount of shuffling from $k = 1$ (no shuffling) to $k = 16$ (full shuffling). Recall that k is the number of registers per set in our design. Figure 3.8(a) shows ρ (y-axis) as we analyze more traces (x-axis). For each k value, we show the average ρ of all 16 weights for two cases: *Continuous line*: the weight with the highest ρ value (ρ_{max}), which is the weight guessed by the CPA attack *Dashed line*: the ρ value of the correct weight ($\rho_{correct}$).

As we use the Pearson Correlation Coefficient, all weights starts with a value of 1. But as we add more traces, we expect $\rho_{correct}$ to stay high while the ρ values for incorrect weights to settle to significantly lower values. This is precisely what we see in the no shuffling case, as the continuous and dashed lines overlap. This means that $\rho_{max} = \rho_{correct}$ and that the attack can

Table 3.2: Time needed (in years) to collect and process enough traces to carry out the attack for a single dimension, for different values of N (columns) and k (rows).

k/N	32	64	128
2	1.91×10^{2}	5.81×10^{7}	7.59×10^{26}
4	3.16×10^{6}	2.10×10^{25}	2.55×10^{63}
8	7.58×10^{13}	5.76×10^{41}	3.33×10^{98}
16	1.27×10^{20}	3.32×10^{56}	2.51×10^{131}
32	8.34×10^{24}	9.37×10^{68}	8.33×10^{160}
64	-	4.02×10^{78}	6.63×10^{185}
128	-	-	1.22×10^{205}

identify the correct weight values in just 30 traces. However, for cases with shuffling, as we analyze more traces, the attack always guesses an incorrect weight as the best guess (i.e., ρ_{max}). The correct weight ($\rho_{correct}$) is consistently lower, giving the attacker no means of identifying this as the correct weight.

Despite this, the incorrect guesses could still be numerically close to the correct weights. To study this, we use the weights obtained with shuffling for one of the networks we study in Chapter 3.5.3, namely *mnist-mlp*. With shuffling, the network achieves a classification accuracy of just 11.7%, compared to the original accuracy of 92.9%. Thus, the weights recovered with shuffling do not provide any useful information to the attacker to steal the network.

Partial Guess Entropy (PGE). In addition to looking at ρ, we also look at how far away the guessed weight is from the correct weight. Recall that the CPA attack generates a list of possible weight guesses, ranked by ρ. PGE [53] is the position of the correct weight in this list of guessed weights; A PGE of 0 means that the attack correctly guessed the weight. Figure 3.8(b) shows the average PGE values for 16 weights as we vary k. As expected, with no shuffling, PGE reaches 0 with just 30 traces analyzed. However with shuffling, PGE values remain high and do not move closer to 0, even with more traces. This shows that analyzing additional traces does not diminish the effectiveness of our technique. We also see that increasing k leads to an increase in the average PGE value. Past 150 traces, we see that the PGE values stabilize in order of k, with $k = 2$ and $k = 16$ having the lowest and highest average PGE values, respectively. This shows that increasing k leads to an increase in the security offered by our design. So far, we have used small values of N and k for clarity. We now quantify the impact of larger values of N and k on the time needed for a successful attack.

3.5.2 Effect on time needed for attack

The increase in number of total permutations is effective as a security measure since it dramatically increases the time needed by the attacker to collect and process enough traces to find the correct weights. Table 3.2 shows the time it would take *(in years)* for an attacker to gather enough traces and process them to recover the weights. We assume an attacker who can gather and process 1000 traces a second, which is similar to the speed of our setup. While the time is relatively short for small values (e.g., ~ 7 days for $N = 32, k = 2$), this rapidly grows into decades and then centuries for larger values of N and k. As the benchmarks we evaluate below contain thousands of weights,

Table 3.3: List of networks evaluated, showing the architecture and overheads (C–Convolutional and F–Fully connected layers).

Network	Architecture	Overhead	
		Software	Hardware
mnist-mlp	$F_{768\times128}$, $F_{128\times10}$	75.82%	1.15%
kws-mlp	$F_{250\times144}$, $F_{144\times144}$, $F_{144\times10}$	59.10%	1.12%
mnist-cnn	$C_{3\times3\times1\times6}$, $C_{3\times3\times6\times6}$, $F_{150\times20}$, $F_{20\times10}$	39.90%	0.19%
har-cnn	$C_{2\times2\times1\times128}$, $F_{5632\times128}$, $F_{128\times128}$, $F_{128\times6}$	271.36%	0.21%
gesture-cnn	$C_{5\times5\times1\times32}$, $C_{3\times3\times32\times64}$, $C_{3\times3\times64\times64}$, $F_{5760\times128}$, $F_{128\times10}$	100.06%	0.14%
ecg-ae	$F_{128\times1024}$, $F_{1024\times1024}$, $F_{1024\times140}$	75.06%	1.17%
seizure-svm	$F_{2854\times179}$	86.74%	0.58%
	Average	101.15%	0.56%

the time needed to carry out the attack would be thousands of years to reverse engineer even a single layer. We further note that the analysis above is for securing one single dimension, such as the weights of a single neuron. Thus, the time needed to reverse engineer a whole network would be cumulative, making it totally untenable to carry out the attack in a reasonable amount of time. This tremendous increase in time needed for the attack is the cornerstone of the security offered by FARO.

3.5.3 System evaluation

In this section, we evaluate the overhead of shuffling using the networks listed in Table 3.3. For each network, we list the layers, the activation function used per layer and the size of each layer. For FC layers, the size is given as input channels × output channels, while for CONV layers it is given as kernel width × kernel height × input channels × output channels. Also, each CONV layer is followed by a 2×2 max pooling layer. The networks are written in C and compiled using ARM GCC 2019.4 compiler, with optimization set to -O3. For performance, we use Thumbulator [110], a cycle accurate simulator for the ARM M0+ CPU. All our networks use 16-bit fixed point values in Q4.11 format.[7] For fully connected and max pooling layers, we shuffle the order of all loops. For convolutional layers, we shuffle input channels, the output channels, input rows and input columns.

Benchmarks. Our benchmarks cover typical networks run on IoT devices. **mnist-mlp** and **mnist-cnn** represent image recognition tasks, which are increasingly popular on IoT devices [164]. **kws-mlp** is an audio keyword spotting network for IoT devices [264]. **har-cnn** classifies users' activities based on accelerometer data [102]. **gesture-cnn** takes camera input and classifies the gesture performed to control an IoT system [261]. **ecg-ae** uses an autoencoder to detect anomalous readings from ECG data [116]. **seizure-svm** processes EKG data to identify the on-set of a seizure, so preventing action can be taken [227].

Software shuffling. For all the benchmarks we study, we see that software shuffling adds a significant overhead, up to 271%. For MLP networks, shuffling takes longer for larger layers, as the list of indices to be shuffled is longer. The overhead for *kws-mlp* is lower compared to the

[7] Our solution also applies to networks that use floating point, such as those shown in CSI NN.

mnist-mlp network, as the former has smaller FC layers. For the CNN networks, benchmarks with more CONV layers have lower overhead. There are two reasons for this: 1) CONV layers have smaller indices which makes shuffling faster and 2) CONV layers require more computation than FC layers. For CONV layers, each weight kernel of size $N \times N$ requires N^2 multiply accumulate (MAC) operations, while FC require a single MAC operation per weight. This higher compute cost amortizes the high cost of software shuffling. However, networks with fewer CONV layers have very high overhead as the first FC layer has a large number of neurons. The overhead from this large FC layer dominates the overhead of software shuffling. In contrast, prior work only shows an 18% overhead for software shuffling [25]. This low overhead is because they only test a very simple MLP network with 15, 10 and 10 neurons per layer. As we evaluate much larger networks, we see significantly higher overheads when using software shuffling.

Hardware shuffling. In contrast to software shuffling, the additional instructions needed for hardware shuffling adds an average of just 0.56% latency overhead. The overhead is higher for the MLP networks as they consist solely of fully connected layers. As we shuffle both dimensions (i.e., neurons and weights per neuron) for FC layers, our technique adds more instructions, leading to greater overhead. We see lower overhead for CNN networks as they spend more time computing convolutional layers. Unlike software shuffling, the overhead of our technique does not scale with the size of layers.

Impact of shuffling on accuracy. Shuffling does not affect network accuracy, as all the operations are still performed, merely in a different order. In contrast, using the weights recovered by the attack when shuffling is used results in a significant loss of accuracy. For example, for mnist-mlp the weights recovered with $k = 16$, result in an accuracy of just 11.7%. It is important to note that shuffling the order of operations does not incur any additional latency due to cache non-locality. Low-power IoT CPUs such as the ARM M0+ do not use caches. Thus, all memory accesses take the same number of cycles to complete.

3.5.4 Area, frequency and power analysis

As mentioned in Chapter 3.4.3, we opt for a design with 16 register per set (i.e., bins). The largest layer in our evaluation is 5760 neurons. We therefore use 10-bit registers, allowing us to support a maximum of 16,384 iterations. With this sizing in mind, we now explore the operating frequency and the area and power overheads of FARO.

We design FARO in Verilog and synthesize it using the Synopsys Design Compiler Version N-2017.09. As IoT devices are typically manufactured using older device technologies [183], we use the TSMCs 65nm (nominal) process technology. For area and delay, we use Cadence Innovus v16.22-s071 and Mentor Graphics ModelSim SE 10.4c. FARO adds just 2.2% area to an ARM M0+ SoC manufactured in 65nm [193]. FARO has an F_{max} of 257.83MHz, which is much faster than the clock speed of IoT devices. Prior works use frequencies ranging from 10MHz to 50Mhz for IoT devices used for ML applications [40, 48, 84, 99]. Thus FARO has no impact on the F_{max} of the overall system. We opt to run our CPU at 24MHz, matching prior work [111]. At this frequency, FARO incurs a 2.22% power overhead, compared to a ARM M0+ CPU [233]. This is in contrast to software shuffling, which, on average, more than doubles the latency and energy cost of computation.

TRNG. We now quantify the randomness required by FARO. Our hardware runs at 24MHz and we use 16 registers per set. As we described in Chapter 3.4.5, FARO produces a new value every 3 cycles. Therefore, we require $24 \times log_2(16) \div 3 = 32$ Mbits/s of randomness. To satisfy this requirement, we use a TRNG which provides up to 86 Mbits/s of randomness [207]. The TRNG adds an additional 0.26% area and 1.06% power overhead, which brings our total overhead to 2.46% area and 3.28% power.

3.5.5 Security of shuffling hardware

We now explore whether FARO can leak any side channel information that an attacker can use to subvert our solution. We use a *formal verification* based approach, which is highly effective in detecting possible side channel leaks. Formal verification has previously identified leaks in a hardware encryption algorithm, which was previously deemed secure based on attacking captured traces [12].

We use the *CocoAlma* [103] tool, which takes a Verilog file as input and searches for possible side channel leaks. *CocoAlma* checks for any variations in latency or power during operation which could potentially serve as a side channel leak. This tool also accounts for hardware leakage effects such as glitches. We analyze FARO using *CocoAlma* and verify that there are no side channel leaks from FARO.

3.6 Broader applicability of FARO

In this section, we outline how FARO can be used for more than just securing ML algorithms against power-side channel attacks. First, we describe two security-critical applications that can be secured using FARO. We then provide an overview of other types of attacks against neural networks running on IoT devices and describe how FARO can also effectively prevent these attacks.

3.6.1 Other applications

Elliptical curve cryptography (ECC). ECC is a public-key cryptography scheme based on elliptic curves over finite fields [26]. ECC encodes keys as coefficients of polynomials. Prior work shows that ECC leaks side channel information, which can be used to recover private keys [47]. Attacks target the *elliptic curve multiplication (ECM)* operation, commonly implemented using the 'double-and-add' method [130]. ECM takes a point p as input and loops over each bit of p; if the bit is 1, ECM performs an *add* operation. Thus, iterations which take longer have a 1 in that bit position. With FARO, we can shuffle the order in which bits are accessed each time, which prevents the attacker from learning which bits are 1. As ECC uses at least 224 bit keys [34], shuffling increases the number of possible permutations tremendously.

Biometric authentication. An emerging use case for IoT devices is for biometric authentication [89]. An example of this is a fingerprint recognition system, such as those commonly used in laptops. Prior work shows that such systems are susceptible to side channel attacks [75]. Specifically, the CPA attack (outlined in Chapter 3.2) can be used to learn each user's stored

fingerprint data [41]. The recognition system is implemented as a set of nested for loops, which can be shuffled using FARO to obscure this side channel.

3.6.2 Other attacks

Floating point timing attack. The difference in time taken by floating point multiplication based on the input values [90] can be used to mount an attack. In the IEEE-754 32-bit floating point format, the smallest number using the normal representation is 1.0×2^{-126}. Numbers smaller than this are called subnormal; operations involving subnormal numbers take much longer than operations using only normal numbers. For example, on an x86 system, $(normal \times normal = subnormal)$ takes 124 cycles, while $normal \times normal = normal$ takes only 10 cycles. During network inference, each $(input \times weight)$ operation has a specific $input$ value which will cause the output to become subnormal. The attacker sweeps the $input$ to find this threshold value and then uses that to learn the $weight$. The attacker can then recover all the weights of the first layer and repeat the process for the other layers. While this attack is limited to networks that use floating point numbers, it requires less equipment as it relies on timing rather than power. However, this attack still requires each operation to occur in the same place in each trace, so the attacker can try multiple $input$ values to find the threshold $input$ value. FARO prevents this attack by randomizing the order of operations, and preventing the iterative search.

Fault injection attacks. The attacks discussed thus far have focused on stealing the model; in contrast, fault injection attacks cause the model to operate in an abnormal way [23, 24]. For example, in the network used for 'chip-and-pin', fault injection can be used to classify a fraudulent transaction as legitimate. Attackers inject 'faults' into the system while it is running the model, forcing it to mis-classify its inputs. Prior work shows a practical attack using lasers to inject faults [23]. To counteract such attacks, techniques have been proposed to detect faults [85, 127]. However, detection techniques incur high overheads and are not 100% accurate. To minimize the chance of detection, the attacker must inject as few faults as possible [83, 265]. Prior work shows that a mis-classification can be forced with just 4 injected faults [82]. However, the attacker must have full knowledge of the model, to determine the exact points where faults must be injected. By shuffling the order of operations, FARO prevents the attacker from determining the exact location for fault injection. The attacker must therefore inject many more faults, and therefore significantly increase the chances of detection.

3.7 Related work

In this section, we present some related work on shuffling to secure encryption algorithms. We also detail related work on *masking*, which is another commonly used technique for preventing side-channel attacks. Finally, we list prior works on securing machine learning in a broader context.

3.7.1 Shuffling

Shuffling was first proposed as a technique to secure AES encryption against side channel attacks [170]. Most shuffling techniques target the 16 S-Box operations performed in AES [243].

We now detail prior work which perform shuffling in software and hardware.

Software. One approach to shuffle the order of operations is to pick a random index to start at each time. As this only requires calculating one random value, it adds significantly less overhead [69, 173, 188]. However, follow-on work shows that this approach does not significantly improve security as it only results in N permutations instead of $N!$ [243]. Another approach is to combine shuffling with inserting dummy instructions to further mis-align the recorded power traces [153]. However, this approach is challenging as the dummy instructions must appear genuine to the attacker or else they can easily remove them from the trace before analysis.

Other approaches perform 'fully shuffling', for securing the S-Box operation of AES running on a low-power CPU [19]. This is done by unrolling the loop which computes the 16 S-Box computations and running these steps in a random order. This technique would be impractical for neural networks due to the larger, arbitrary number of neurons and weights in neural networks. FARO solves this problem and allows shuffling arbitrary number of neurons and weights, using dedicated hardware.

Hardware. Shuffling in hardware has also been implemented for AES on FPGA [213, 248]. Dhanuskodi et al. design an ASIC for AES encryption, which employs shuffling [49]. Techniques that combine hardware and software to perform AES shuffling have also been proposed [79]. Patranabis et al. detail hardware to perform shuffling in hardware in two rounds instead of one [213]. All these works target AES encryption and cannot be extended to support shuffling arbitrary number of operations, which is necessary for shuffling neural networks. In contrast, our hardware is able to scale to arbitrary number of operations.

Other uses. Adding hardware for shuffling has also been proposed for other encryption algorithms such as elliptic curve cryptography [34] and lattice-based cryptography [38]. These approaches are also restricted to shuffling 2^N iterations, while FARO supports shuffling any number of iterations. Shuffler [252] and Morpheus [76] employ shuffling to protect against code reuse attacks, while we defend against side channel attacks.

3.7.2 Masking

One popular technique to obfuscate side-channels is to *mask* secret data by *splitting* this data into several parts and operating on each part separately. Mathematically, the secret information s is split into d parts (or 'shares') $s_1, s_2, ...s_d$ such that $s_1 \oplus, s_2 \oplus ... \oplus s_d = s$. Thus, for an attacker to recover s, they must first recover all d shares. As the d shares can be distributed anywhere in the power trace, the attacker must analyze the entire trace to try to identify where in the trace each share is located. This process makes masking effective at improving security against side-channel attacks.

Masking has been extensively studied for securing encryption algorithms such as AES [140], Saber [145], and Midori64 [81]. Masking has also been applied to CPUs to prevent side-channel attacks but incur a 141× latency overhead [8]. Similar to our approach, Dubey et al. modify a RISC-V CPU to mask operations during network inference, but add 2× latency overhead [58], compared to just 0.56% added by FARO.

Techniques to secure neural networks accelerators by masking have also been proposed, although these techniques impose significant latency (up to 2.8×) and area (up to 5.9×) over-

heads [56, 57]. Maji et al. propose a masking-based neural network accelerator to prevent power side-channel attacks, which adds 1.4× latency and 1.64× area overhead and only targets fully connected layers [166]. In contrast, FARO adds just 0.56% latency and 2.46% area overhead to an ARM M0+ SoC.

Another solution to obscure side channel leakage due to memory access patterns is oblivious RAM (ORAM) [88]. However, ORAM cannot be used to hide power side channels, which is the focus of our work [44]. Also, ORAM imposes a large 100× overhead, compared to just a few percent for our technique [2].

3.7.3 Machine learning side channel security

Attacks against ML algorithms using cache side channels have also been proposed [115, 256]. These attacks leverage the difference in cache access timing to infer information. However, as IoT devices typically lack caches, such attacks do not apply to them. Similarly, the memory access pattern of neural network accelerators has also been used as a side-channel to recover information [119]. The authors are able to reverse engineer the network architecture from the memory access patterns observed during inference. This attack does not apply to low-power embedded systems, where all memory accesses take the same number of cycles. Recent work proposes a defense against such an attack that employs shuffling, but is limited to a small subset of memory accesses to obscure only the boundaries between layers [161].

3.8 Conclusion

In this chapter, we propose a technique to efficiently and securely shuffle the order of operations in an ML model running on an IoT device. We show that software shuffling (proposed in prior work) leaks information which can be used to obviate the security benefits of shuffling. Leveraging this, we detail a new attack against software shuffling which can be used to learn the exact values being shuffled from a single power trace. To overcome the short-coming of software shuffling, we propose FARO, which adds a hardware functional unit within the CPU. FARO uses a novel counter-based approach, to effectively shuffle the large, arbitrary sizes of neural network layers. FARO adds just 0.56% latency overhead, compared to over 100% in the case of software shuffling. We show that FARO effectively secures the weights of neural networks and can also be used to secure other applications. FARO adds just 2.46% area and 3.28% power overhead, without itself leaking any side channel information.

4

AESIR: DISTRIBUTION AGNOSTIC NOISE INJECTION IN MACHINE LEARNING HARDWARE

In this Chapter, we consider the security and privacy of edge ML accelerators. We have seen a tremendous rise is the number of edge accelerators targeting ML [192]. However, there has been a lack of work that studies such devices from the safety and privacy perspective. As they are designed to mimimize power consumption, edge ML accelerators typically lack a CPU [36, 37, 86, 146, 211]. Thus, the accelerator is responsible not only for performance but also for security. In this Chapter, we examine a common requirement for running security-centric ML algorithms that current ML accelerators do not provide. Specifically, these algorithms require *random numbers*, sampled from specific distributions. Examples of such algorithms, and the type of random values they require are:

Differentially-private ML : Differential Privacy (DP) is a technique that allows for operations to be performed on large collections of private data, while protecting the privacy of each individual's data [4]. Differentially Private ML (DP-ML) extends the privacy-preserving guarantees of DP to machine learning models. Models trained on private user data can 'memorize' specific examples and can leak sensitive user data [72]. Thus, to ensure the integrity of ML models trained on private data, DP-ML adds Gaussian or Laplace random values to the activations of each layer during training. We further elaborate on DP-ML in Chapter 4.1.1.

Adversarially robust CNNs : As described in Chapter 2.2, adversarial attacks are a major risk to ML systems. Many techniques which aim to improve robustness against such attacks require random values sampled from a Gaussian distribution [65, 109, 128, 268].

One option for supporting these algorithms is to include dedicated hardware for noise[1] generation. However, as we show in Chapter 4.2.1, existing approaches to do so impose high overheads which are untenable in resource constrained edge devices. Also, adding support for different distributions (e.g., Gaussian or Laplace) and varying dimensionality (i.e., univariate or multivariate) further increases these overheads. Finally, hardware noise generation can also leak side-channel information [186]; popular methods to produce both Gaussian- and Laplace-sampled noise are susceptible to timing side channel attacks [131].

[1] We use the terms 'noise' and 'random values' interchangeably throughout this chapter.

To efficiently and safely enable security-centric ML algorithms on edge devices, we propose AESIR.[2] Instead of computing noise values directly in hardware, AESIR pre-computes and stores noise values in memory ahead of time. During runtime, we *randomly sample* from these stored points to produce the noise we need. AESIR avoids the overhead of noise-generation hardware in favour of storing pre-computed noise points in plentiful off-chip memory. AESIR adds significantly less overhead compared to dedicated hardware for noise generation. Thus, AESIR allows ML accelerators to support a wide range of ML algorithms that require noise sampled from a variety of distributions. Furthermore, AESIR does not suffer from the vulnerability to side-channel attacks as dedicated noise generation hardware.

This chapter is based on work published in:

> K. Ganesan *et al.*, "Dinar: Enabling distribution agnostic noise injection in machine learning hardware," in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2023, pp. 38–46

Contributions

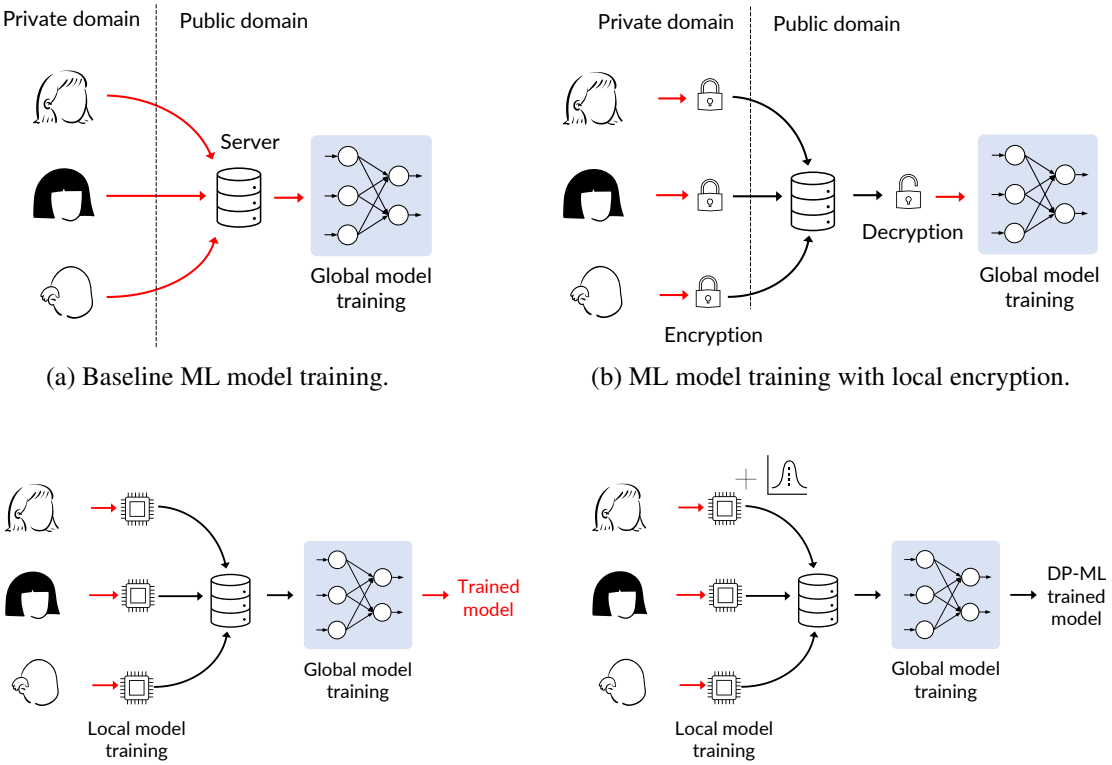
In this chapter, we make the following contributions:

- We observe that current edge ML accelerators have no way of generating noise from specific distributions and therefore are unable to run many security-critical ML algorithms.
- We explain why current hardware-based noise generation approaches suffer from high overheads and are insecure.
- To efficiently and safely produce random values, we propose AESIR, lightweight hardware modifications to ML accelerators to enable noise generation. AESIR pre-computes and stores random values in off-chip memory and **randomly samples** from these stored values during runtime.
- AESIR enables crucial algorithms such as differentially private ML and adversarially robust CNNs, with $< 0.5\%$ area and energy overheads. Compared to prior approaches, AESIR adds $23\times$ lower area and $40\times$ lower energy, while avoiding the security issues with dedicated noise-generation hardware.

4.1 Background and related work

We now provide some background on differentially private ML, which is one of the main algorithms which requires random noise (Chapter 4.1.1). We then describe how techniques for improving adversarial robustness also require random noise (Chapter 4.1.2). Next, to understand how we implement AESIR, we provide an overview of the architecture of typical edge ML accelerators (Chapter 4.1.3). Finally, we outline existing hardware approaches for generating Gaussian and Laplace noise in hardware (Chapter 4.1.4).

[2] AESIR: Architecturally Efficient Stochasticity Injection for Robustness



(a) Baseline ML model training.

(b) ML model training with local encryption.

(c) Federated learning.

(d) Differentially Private machine learning.

Figure 4.1: Different scenarios for training a ML model on private user data. Red lines show movement of data which is not secured, while black lines show movement of secure data.

4.1.1 Differentially Private ML

In this section, we begin by outlining the threat model that DP-ML is designed to protect against. We then describe differential privacy before detailing how differential privacy is used in conjunction with ML.

4.1.1.1 Threat model

We consider training a single ML model using private data from several users. We show different scenarios to achieve this, and the different security vulnerabilities in each case. We depict data movement in each figure with arrows; red arrows indicate the data is not protected and vulnerable to theft. We start with the base case (Figure 4.1a) where private data from several users is sent to a central server to train a model. In this case, data is sent 'as-is' and is vulnerable during transmission.

To secure data during transmission, Figure 4.1b shows the case where data is first encrypted before transmission. Encryption protects the data during transmission (shown with black arrows in the figure) and only requires modest computational capability on the user side. However, on the server side, this scenario requires the server to decrypt data from every user before training the model. For systems with many users, this can add significant overhead to the server [169]. In addition, decrypted data from every user is stored on the server during training, making the server a 'single point of failure'. If an attacker gains access the server, they can obtain unencrypted data

from every user in a single location. Some approaches (e.g., homomorphic encryption) aim to allow the server to directly train on the encrypted data. However, homomorphic encryption slows down execution by several orders of magnitude [67, 154].

Federated learning (Figure 4.1c) avoids having to store data from every user on the server [169]. In this approach, each user trains a local ML model and then shares just the updates with the central server. The server then trains a global model with all the local updates and shares this model back with all the users. However, prior work shows that even the trained model can leak information [72]. Thus, we require a means of enabling federated learning while protecting user privacy. One approach to remedy this is differentially private ML, which we describe next.

4.1.1.2 Different Privacy

Differential privacy (DP) leverages the idea that a user's privacy is guaranteed if their data is not in the dataset at all. DP therefore gives each user the same privacy that they would get from having their data removed from the dataset. Dwork et al. [59] demonstrate that random noise can be added to each user's data to mimic the effect of removing one user's data from the data set. DP is now used by major companies such as Apple [9] and Facebook [197] and even the US Census Bureau [1] to secure private user data.

The most commonly used is random noise sampled from a Laplace distribution [80], which yields a strict ε-DP guarantee and is termed *pure differential privacy*. ε is a tunable parameter to control the trade-off between privacy and accuracy [152]. Another option is to add noise from a Gaussian distribution – termed *approximate differential privacy* – which provides the looser (ε, δ)-DP guarantee [80]. In this case, δ is a small value which quantifies the risk of privacy loss from using Gaussian noise. The Laplace mechanism is preferred for smaller datasets, while Gaussian is preferred for larger datasets [4].

Algorithm 3: Differentially private SGD.

Inputs: Examples $(x_1 \ldots x_N)$, timestep (T), weights (w), minibatch size (B), loss function $(\mathscr{L})$, noise variance (σ), max gradient norm (C), learning rate (η)
Initialize model weights before timestep 0: w_0
for $t \in [T]$ **do**
    Randomly sample a minibatch B.
    foreach $i \in [B]$ **do**
        ▷ Compute gradients
        $g_t(x_i) \leftarrow \nabla_{w_t} \mathscr{L}(w_t, x_i)$
        ▷ Compute L2 norm
        $n_i = \|g_t(x_i)\|_2$
        ▷ Clip gradients
        $\overline{g}_t(x_i) \leftarrow g_t(x_i) / max(1, \frac{n_i}{C})$
    end
    ▷ Add noise and aggregate
    $\widetilde{g}_t(x_i) = \frac{1}{B}\left(\sum(g_b^c + D(0, \sigma \cdot C)\right)$
    ▷ Perform SGD
    $w_{t+1} \leftarrow w_t - \eta \widetilde{g}_t$
end

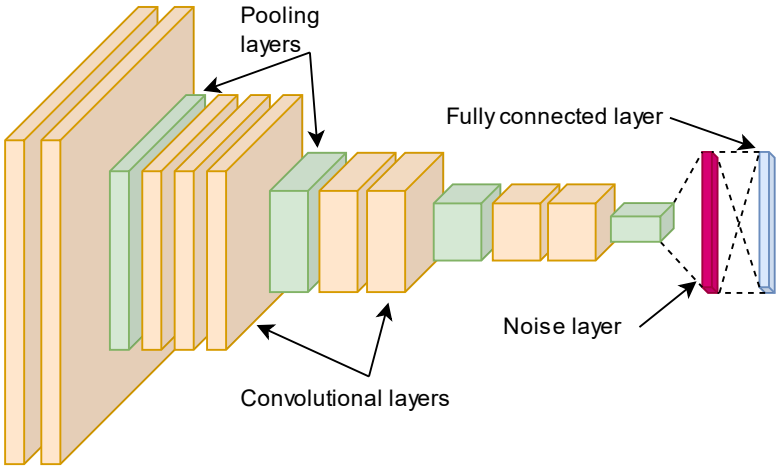


Figure 4.2: Architecture of WCA-Net, showing the noise layer before the final fully connected layer of the network.

4.1.1.3 Differentially Private ML

Differentially Private ML (DP-ML) extends the privacy-preserving guarantees of DP to machine learning models. Figure 4.1d shows this, depicting the addition of noise during local training on the edge ML accelerator. A popular technique for training DP-ML models is to modify the standard Stochastic Gradient Descent (SGD) algorithm. This new algorithm (called DP-SGD) is outlined in Algorithm 3). Similar to regular SGD, DP-SGD first computes the gradients with respect to the loss function (Line 6). After this, DP-SGD computes the L_2 norm of the computed gradients (Line 7) to perform gradient clipping (Line 8). Then, DP-SGD adds noise to each gradient value before the final aggregation step (Line 10). This final step of adding noise to each gradient is crucial for the certifiable security guarantee provided by DP-ML.

4.1.2 Adversarial robustness

We now explain how AESIR can enable the execution of adversarially-robust models on edge devices. As described in Chapter 2.3.1, a widely studied technique to improve robustness is to add 'noise' (i.e., random values sampled from a specified distribution) to the network [109, 128, 149, 160]. We demonstrate how AESIR can be used to add noise using a recently proposed technique: **WCA-Net** [65]. We choose WCA-Net as it achieves state-of-the-art improvements in model robustness without adversely affecting classification accuracy.

As shown in Figure 4.2, WCA-Net adds a noise layer before the final fully-connected layer of the network. WCA-Net injects randomly sampled *anisotropic multi-variate* Gaussian noise, during both training and inference. This is coupled with a modified loss function which aligns the weights of the final layer with the injected noise. This causes the final layer to better 'separate' benign and adversarial inputs so that a small perturbation in the input does not cause a large change in the final layer output. With this approach WCA-Net achieves state-of-the-art performance by injecting noise at just one point in the network.

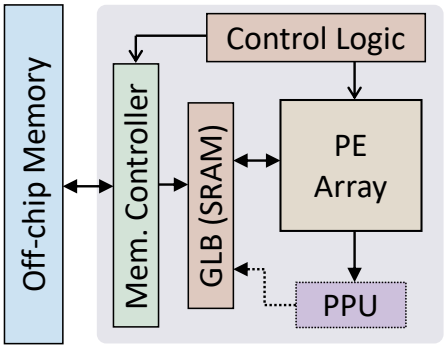


Figure 4.3: Overview of a generic edge ML accelerator.

4.1.3 CNN accelerators

In this section, we provide a brief overview of CNN accelerators. We also describe prior work which aims to improve the security of these accelerators.

4.1.3.1 Architecture

We describe a typical ML accelerator as shown in Figure 4.3. For inference, accelerators store the trained weight values in off-chip (DRAM) memory. Accelerators run one layer at a time, with the weights and inputs for that layer being moved from off-chip memory to the on-chip global buffer (GLB). The weights and inputs are then sent to the processing elements (PEs). The PEs are connected as a systolic array [86, 134] or using a network-on-chip [36, 37, 146].

Inference vs. training. We consider the application of our techniques to both training and inference. ML accelerators for training must support back propagation of the gradients to adjust the weights. Inference and training accelerators differ in the datatypes supported; inference is commonly done with fixed-point formats such as int8 [134] or int16 [36, 55], which results in significant area, energy and latency savings, while training uses floating point such as bfloat16.

System integration. To meet strict area and energy constraints, edge ML accelerators are typically deployed 'standalone', without a CPU [36, 37, 86, 146, 211]. The designs we use as our baseline for evaluating training and inference – Eyeriss [36] and DiVa [211] – both lack a CPU. Since edge ML accelerators typically run a single network for long periods of time, they do not need the flexibility of a tightly-coupled CPU. They are instead configured, when required, using a scan-chain [36], a configuration bit-stream [86] or using on-chip control logic [146]. We therefore consider efficient noise addition in accelerators without CPUs.

4.1.4 Sampling distributions in hardware

We now explore existing approaches for producing random values from Gaussian and Laplace distributions, on-the-fly. First, we describe our notation for distributions used throughout our work. Gaussian and Laplace distributions are typically characterized by their mean (μ) and scale (σ). The Gaussian distribution is commonly characterized using it's variance (σ^2), which is the square of the scale. However, for consistency with the Laplace distribution, we use the scale for both distributions. We also always consider distributions with a mean of 0 as none of the

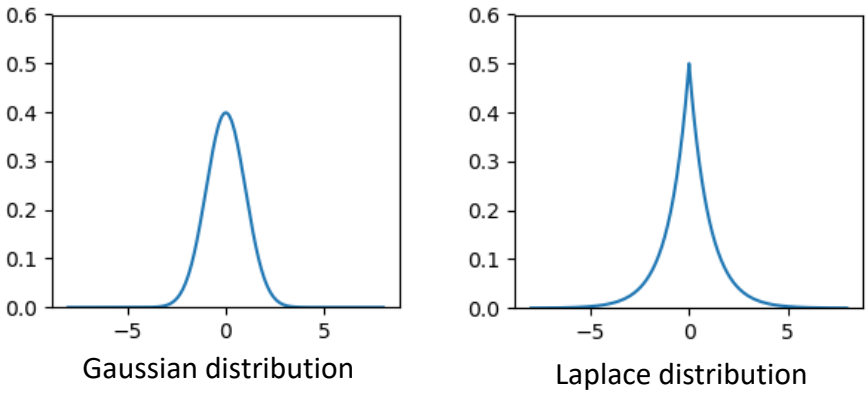


Figure 4.4: Gaussian and Laplace probability distribution functions.

applications we study uses a non-zero mean. Therefore, we denote the Gaussian and Laplace distributions as G^{σ} and L^{σ}, respectively.

A hardware random number generator (RNG) produces N-random bits, following a *uniform distribution* $U \in [-2^{N-1}, 2^{N-1}-1]$. We must then convert this uniformly-distributed value to the required Gaussian or Laplace distribution. A common approach, which works for any distribution, is the *inversion* method. If we wish to generate values for a random variable X, we start with the distribution's cumulative distribution function (CDF) (i.e., F_X) and calculate the inverse CDF (i.e., F_X^{-1}). Given a uniformly-distributed random number $u \in [0,1]$, $F_X^{-1}(u)$ is a random variable with the required distribution. For example, for the Laplace distribution, the CDF is

$$F_L^{\mu,\sigma} = \frac{1}{2\sigma} exp\left(\frac{-|x-\mu|}{\sigma}\right)$$

Applying the inverse, we obtain

$$F_L^{-1}(u) = \mu - \sigma sign(u-0.5) ln(1-2|u-0.5|)$$

Thus, we can convert a uniformly-distributed value, produced by an RNG, to a Laplace-distributed random number by implementing this inverse CDF in hardware.

Gaussian Noise: Unfortunately, the Gaussian distribution does not have a closed-form inverse CDF function. Thus, we cannot use the same approach we did for the Laplace distribution. We opt for a popular technique, namely the *Box-Muller* transform [22]. We elaborate further on this technique in Chapter 4.3, as we implement this design as a baseline to compare against AESIR. However, we later show that this design still requires expensive hardware blocks to implement. This motivates our design of AESIR, which we describe next.

4.2 AESIR

We motivate and describe AESIR, our technique for enabling efficient noise injection in ML accelerators. We begin with the challenges faced by existing approaches which generate noise directly in hardware. We then provide an overview of the hardware required for AESIR. Finally, we explain how AESIR overcomes the challenges of existing approaches.

4.2.1 Challenges

We now describe the difficulties associated with producing the noise required by security-centric ML algorithms. For ease of later referencing, we number each of these from $\mathbb{C}1$ to $\mathbb{C}5$. The first four–$\mathbb{C}1$ to $\mathbb{C}4$–apply to DP-ML, while $\mathbb{C}5$ is relevant for adversarial robustness.

$\mathbb{C}1$**: Altering the scale of the produced noise.** DP-SGD (described in Chapter 4.1.1) uses the same noise scale (σ) for all layers of the network. As we explained in Chapter 4.1.4, noise hardware typically produces $G^{0,1}$ or $L^{0,1}$ noise. However, DP-ML techniques add noise with varying scale values to the activations of all layers in the network [27, 210]. To get $G^{0,\sigma}$ or $L^{0,\sigma}$ noise, we must perform an additional multiplication operation, as shown in Figure 4.7a. For Gaussian noise, this requires two multipliers, as we produce two Gaussian random values at a time.

$\mathbb{C}2$**: Floating-point noise.** As described in Chapter 4.1.3, accelerators used for training typically use floating-point values. However, existing approaches produce *fixed-point* values. These fixed-point values must first be converted to floating point, which requires additional hardware.

$\mathbb{C}3$**: Continuous noise.** To provide certifiable privacy guarantees, DP requires the added noise to follow an 'ideal' distribution. Every possible value from a distribution must be a possible output without any 'gaps'. However, the transforms described above do not satisfy this condition [186]. Assuming the uniform random numbers are evenly spread out – which is the case when using a true random number generator, the transformed distributions are then unevenly spaced, with more values around 0 and fewer values with larger magnitudes. Prior work shows how this can be exploited by an attacker to undermine the guarantees offered by differential privacy. The solution proposed by the authors – called 'clamping' – would incur additional overheads to prevent this [186].

$\mathbb{C}4$**: Timing side channel free noise.** Jin et al. show that techniques to sample from Gaussian and Laplace distributions also suffer from timing side channels [131]. The time taken to produce a sample leaks the magnitude of the noise. With a success rate of over 90%, this timing attack effectively subverts the security of differentially private systems.

$\mathbb{C}5$**: Anisotropic multivariate noise.** Producing the noise required by WCA, the adversarial robustness technique we study, is also non-trivial for noise generation hardware. As we explained in Chapter 4.1.2, WCA requires multi-variate Gaussian noise. The univariate distribution is described by two scalar values (i.e., the mean μ and scale σ). However, the multi-variate distribution ($\vec{G}$) is described by a vector of means $\vec{\mu}$ and a covariance matrix Σ. Furthermore, WCA requires *anisotropic* multi-variate noise.[3] To obtain K-dimensional anisotropic noise, we cannot simply use K separate univariate noise values. Instead, we must multiply these K independent noise values with the covariance matrix (Σ) to generate the required noise vector $\vec{K}$.

With these challenges in mind, we describe AESIR, our noise injection approach which overcomes these challenges.

[3] For anisotropic noise, the covariance matrix (A) must be *symmetric* (i.e., $A = A^T$) and *positive semi-definite* (i.e., $b^T Ab \geq 0$, for $b \neq 0$).

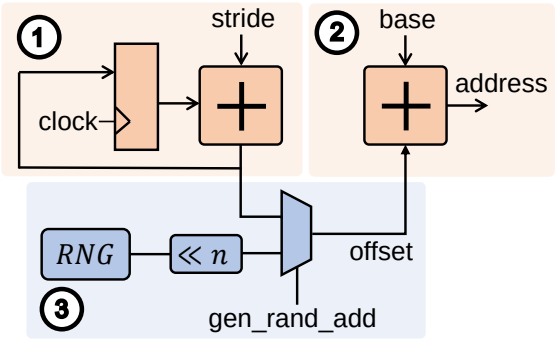


Figure 4.5: AESIR modifications to the address generation logic.

4.2.2 High level overview

AESIR overcomes the challenges described above by **not producing noise directly in hardware**. Instead, we pre-compute the required noise values ahead of time and store them in plentiful off-chip DRAM memory. During runtime, our modified hardware simply reads a random value from this stored list to produce the required noise. AESIR only requires lightweight modifications to the memory controller and **imposes no other limitations on the architecture of the ML accelerator.** Specifically, AESIR modifies the address generation logic inside the memory controller (Figure 4.3).

Figure 4.5 shows the hardware for address generation and the modifications made to support AESIR. The region labelled ② calculates the addresses sent to DRAM as a 'base' plus an 'offset'. The 'base' indicates the starting address in memory of the item being read (e.g., the weights of a particular layer). The region marked ① increments the 'offset' by a fixed 'stride' every cycle. As layer parameters – such as weights and inputs – are much larger than the size of the DRAM bus, we must perform multiple read operations to load all the values from DRAM. To support this, the 'offset' is incremented by the 'stride' value during each cycle to load the next set of values from DRAM.

We modify this hardware to support reading random values instead of the stride during each cycle. Our additions are shown in the region marked ③. We add a random number generator (RNG) to select a random instead of fixed value from memory. A $2:1$ mux selects between the original offset and our randomly calculated offset. The `gen_rand_add` signal selects between these two modes. We also add the address where random values are stored in the chip's control logic, so that the correct 'base' address will be provided for reading random values. We also add logic to perform bit-shifts, which is required for the WCA algorithm, as we elaborate in Chapter 4.2.3.1. We now explain how we hide the latency of DRAM reads by loading noise values while the previous layer is still running.

4.2.3 Scheduling DRAM reads

To easily integrate with the scheduling schemes of existing accelerators, we frame noise addition as running a 'noise layer'. The inputs to this noise layer are the activations from the previous layer, while the 'weights' of this noise layer are the random values we wish to add. Consider the case of a noise layer being run between layers N and $N+1$ of a network. The weights for layer

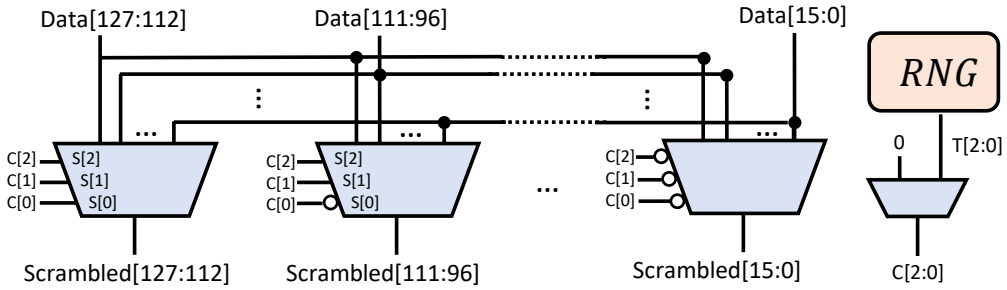


Figure 4.6: Hardware to scramble values read from DRAM. For clarity, some muxes and data lines are omitted.

N have been read from DRAM and are stored in the weight buffer. Once the input for layer N is ready, that layer is run on the PE array.

Since the weights for layer N have been dispatched, we can now read the random values from DRAM and store them as 'weights' in the weight buffer. Depending on the algorithm, we can have one (e.g., WCA) or many (e.g., DP-ML) noise layers per network. Once layer N has finished, we schedule the noise layer to add the random noise values to all the activations. The noised activations are then ready to be used as the inputs to layer $N+1$.

4.2.3.1 Support for WCA

We now describe how AESIR supports both algorithms we evaluate, starting with WCA. For WCA, we require noise values which must be used **in order** every time; we refer to these values as a **noise vector**. WCA requires either 32 or 256 values per vector. As DRAM is typically much wider than the datatype used, reading a single vector requires multiple DRAM reads. For example, consider a 128-bit DRAM bus and an int8 datatype where we store 16 values together. Thus, 32 values will be stored across two addresses while 256 values will be stored across 16 addresses. Therefore, when reading random noise vectors, we need a way to index to the starting address of each vector.

To do so efficiently, we add an n-bit shifter to our hardware, as shown in Figure 4.5. We use bit-shifts to calculate $2 \times R$ and $16 \times R$, where R is the output of the RNG. This allows us to always read an entire noise vector that is either 32 or 128 values in size each time. When we do not wish to use the shifter, we simply set $n = 1$ (as is the case for DP-ML).

4.2.4 Support for DP-ML

For DP-ML, we use univariate noise and thus do not need to do multiple DRAM reads. However, for each individual DRAM read, we always use that set of values in the same order. For the 128-bit DRAM bus and a 16-bit datatype we use in our evaluation of DP-ML (Chapter 4.4), 8 values are used in the same order every time. Although this ordering is necessary for WCA, for DP-ML, this can potentially diminish the security of our approach. To prevent the possibility of any information leakage from using univariate points in order every time, we add hardware that scrambles the order of these values each time they are read.

Scrambling hardware. Figure 4.6 shows our scrambling hardware. We require $128/16 = 8$ multiplexers so that each 16-bit value can be placed in any position in the final 'scrambled'

128-bit output. We need a 3-bit TRNG (i.e., $log_2(8)$) to select a random permutation each time. The TRNG output is connected to the select signal of each mux, with varying negation (shown with the white circle on the mux select lines). This ensures that each mux selects a unique 16-bit value from the input. This scrambling produces $8! = 40,320$ possible scrambled orders. Coupled with the large number of overall values we store in DRAM (discussed in Chapter 4.4), this results in a huge number of possible random values. This makes it untenable for an attacker to learn the noise values that were added to activations to subvert the security of AESIR.

4.2.5 Benefits of AESIR

We now explain how AESIR addresses each of the challenges faced by prior approaches (Chapter 4.2.1).

- $\mathbb{C}1$: As DP-ML adds noise from a distribution with a specific scale (σ) for every layer, AESIR simply stores values from this specific distribution and avoids any additional multiplication operations.
- $\mathbb{C}2$: AESIR can produce noise from any datatype without needing additional hardware to convert between datatypes.
- $\mathbb{C}3$: By producing noise points ahead of time, we can produce noise sampled from an 'ideal' distribution without missing any values and compromising the privacy guarantee offered by DP.
- $\mathbb{C}4$: As AESIR only performs memory read operations to get new noise values, there is no variation in time depending on the value being read. Therefore, we naturally obtain a constant time implementation, making AESIR immune to timing side-channel attacks.
- $\mathbb{C}5$: For WCA, we once again pre-compute and store several noise vectors and randomly choose one during runtime. This avoids the matrix-multiplication operation to convert k independent noise points into a noise vector.

While AESIR overcomes these challenges, the hardware for generating noise in hardware (described in Chapter 4.1.4), fails to do so. Next, we describe our hardware for on-chip noise generation, which we use as a baseline for comparing against AESIR.

4.3 Baseline on-chip noise generation hardware

In this section, we describe our baseline implementation, which we call NoiseGen. To support both DP-ML and WCA, NoiseGen must produce both Gaussian and Laplace noise. For Gaussian, we choose the design proposed by Lee et al. [150], which implements the Box-Muller transform. We opt for this design as the Box-Muller transform is considered the best approach to balances accuracy and hardware usage [168]. The Box-Muller transform (Equation 4.1) converts two uniformly distributed random numbers U_0 and U_1 to two Gaussian random numbers G_0 and G_1

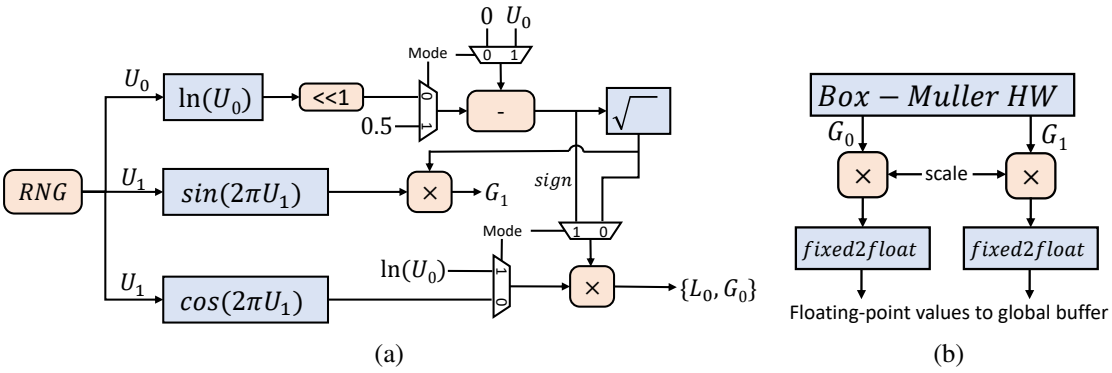


Figure 4.7: (a) Hardware to produce $G^{0,1}$ and $L^{0,1}$ random values. (b) Additional hardware to produce $G^{0,\mu}$ and $L^{0,\mu}$ values and convert them to float (if needed).

$$G_0^{\sigma} = \sigma(\sqrt{-2ln(U_0)} \cdot sin(2\pi U_1))$$
$$G_1^{\sigma} = \sigma(\sqrt{-2ln(U_0)} \cdot cos(2\pi U_1)) \tag{4.1}$$

For the Laplace distribution, we use Equation 4.2 from Choi et al. [40], which converts U_0 and U_1 to a single Laplace random number $L^{\mu,\sigma}$.

$$L^{\sigma} = \sigma(sign(U_0 - 0.5) \cdot ln(U_1))) \tag{4.2}$$

Hardware implementation. Figure 4.7a shows the block diagram of our implementation of the design of Lee et al. [150]. Blocks in blue calculate transcendental functions, while blocks in orange (with rounded corners), compute basic math operations (e.g., multiplication, subtraction). All transcendental functions (i.e., $ln()$, $sqrt$, $sin()$ and $cos()$ are implemented using lookup tables, where the coefficients are determined using Chebyshev series approximations [226]. To reduce the table size, this design employs range reduction to first transform the input to each table into a small range of values [226]. A key caveat with this hardware is that – similar to other hardware techniques for noise generation – it always produces $G^{0,1}$ noise.

Generating Laplace noise. The hardware proposed by Lee et al. only produces Gaussian random values. However, as we explained in Chapter 4.1.1, DP-ML sometimes requires Laplace random values. To enable this, we make minor modifications to the hardware shown in Figure 4.7a to also produce Laplace random values, based on the formula from Choi et al. [40]. Since some hardware blocks are shared between Equations 4.1 and 4.2 (i.e., $ln(U_0)$ and a multiplier), we add muxes to switch the inputs to those blocks. We use the *Mode* input to switch between generating Gaussian ($Mode = 0$) and Laplace ($Mode = 1$) noise. In Gaussian mode, we produce two Gaussian random numbers, while in Laplace mode, G_1 is ignored. To maximize throughout, we fully-pipeline our design to produce 1 random value per cycle, after an initial start-up latency of 16 cycles for Gaussian mode and 8 cycles for Laplace mode.

As described in Chapter 4.2.1, generating noise directly in hardware faces a number of challenges. In this section, we describe how we modify the hardware in Figure 4.7a to address

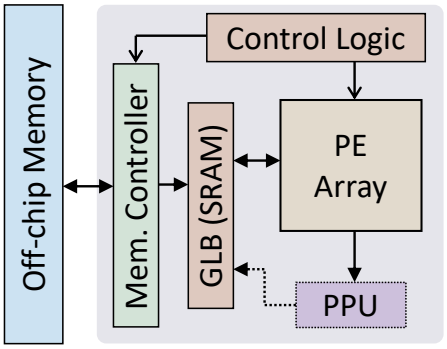


Figure 4.8: Architecture of the *DiVa* [211] accelerator.

two of these challenges. We do this to enable a fair comparison against AESIR in our evaluation. We refer to the final hardware, shown in Figure 4.7b as NOISEGEN.

$\mathbb{C}1$**: Supporting multiple scale factors.** As described in Section 4.2.1, DP-ML requires algorithms with varying scales (e.g., $G^{0,\sigma}$ for Gaussian). Recall that the hardware in Figure 4.7a only produces $G^{0,1}$ random values. Therefore, we add two additional multipliers in Figure 4.7b to produce $G^{0,\sigma}$.

$\mathbb{C}2$**: Floating-point noise.** NOISEGEN produces fixed-point random values, which must be converted to floating point values using additional hardware. We pipeline our `fixed2float` hardware, which requires 16 cycles to convert a 16-bit fixed-point to a 16-bit floating-point value. Thus, the start-up latency of NOISEGEN is 32 and 24 cycles, for Gaussian and Laplace modes, respectively.

To save energy, we also power gate NOISEGEN when it is not being used. Finally, to match the operation of AESIR, we store the noise values produced by NOISEGEN in global memory until needed. Next, we present our evaluation of AESIR, for DP-ML.

4.4 AESIR for DP-ML

We now evaluate AESIR when used to add noise for DP-ML. We begin with our methodology for evaluating NOISEGEN against AESIR for this algorithm. We then detail the overheads of AESIR and NOISEGEN when used to generate noise for DP-ML.

4.4.1 DP-ML Methodology

Accelerator description. In this section, we focus on the accelerators targeting training. Our design is based on the *DiVa* [211] architecture, shown in Figure 4.8. However, *DiVa* is designed for datacenters, while we target edge applications. Therefore, we opt for a smaller version of *DiVa*, suitable for low-power edge scenarios. The post-processing unit (PPU) proposed in *DiVa* can perform a sum-reduce of 128 values, to match their 128×128 PE array. Instead, the PPU in our design can operate on 16 values at once. We assume a DRAM bus width of 128 bits, similar to prior work [14, 68].

We evaluate two flavours of the training accelerator to compare the two approaches for noise addition. The first (`NoiseGen-T`) generates noise directly in hardware, using NOISEGEN.

Table 4.1: List of models evaluated. For each, we show the dataset used and the storage space needed for the model. We also provide a short name we use in our evaluation.

Model	Dataset	Model size (MB)	
		int8	float16
PreActResNet-18 (P18)	CIFAR-10	11.18	22.36
	CIFAR-100	11.32	22.64
WideResNet-32 (W32)	CIFAR-10	46.18	92.36
	CIFAR-100	46.35	92.70
VGG-16 (V16)	CIFAR-10	14.75	29.50

This variant also includes hardware to convert from fixed-point to floating-point (Chapter 4.3). The second flavour (`AESIR-T`) implements AESIR, to read random values from DRAM. This variant includes hardware to scramble values read from DRAM. As we use a 128-bit DRAM bus and a 16-bit datatype, we read 8 values from DRAM per cycle. Thus, our scrambling hardware uses 8 muxes.

Modelling accelerators. We model our accelerator using Accelergy [255]. Accelergy supports components commonly used in accelerators such as multiply-accumulate units (MACs), SRAM for global memory, networks-on-chip, and the interface to DRAM. We implement additional hardware in Verilog including: the PPU and the hardware shown in Figs. 4.6 and 4.7b. We then perform synthesis using the Synopsys Design Compiler 2017.09 for the TSMC 65nm (nominal) process. We clock our design at 200MHz, matching prior work [36, 245].

Test setup. For training differentially private ML models, we use the Opacus [259] library, which adds DP-ML support to the PyTorch [212] framework. Table 4.1 shows the networks we study. We evaluate three models using two datasets, similar to prior work [74].

4.4.2 Overheads

We now analyze the overheads of AESIR by comparing `NoiseGen-T` and `AESIR-T` against our baseline implementation of *DiVa*. We begin by detailing the overheads for both which are similar, such as latency and global memory overhead. We then detail the area and energy overheads for each flavour separately, as they vary significantly.

Global memory. For both flavours, we store noise points in the on-chip global memory (GLB) until they are needed. This raises the possibility that we may not have enough space in the GLB to store the noise points, without evicting other data. We analyze this using Accelergy, which outputs the utilization of the GLB per layer. We see that there is no layer in any of the networks we evaluate where the GLB is more than 90.5% full. Thus, we always have at least 12 kB of space in the GLB for storing noise points. This allows us to read and store over 6000 noise points from DRAM per layer. Thus, for the networks we evaluate, storing noise points does not lead to any global memory contention.

Latency. As both flavours read noise values while the previous layer is running, both incur a minimal latency overhead of just 0.49% for all the networks we evaluate. This minimal increase comes from the extra latency of adding noise after the penultimate layer. The overhead is minimal for both flavours as they are both designed for low-latency operation. However, enabling low

Table 4.2: DRAM storage footprint for varying amounts of noise points stored.

Number of noise values in DRAM	Storage required (KB)	Footprint (%) over smallest model
65536	128	0.56
131072	256	1.11
262144	512	2.23
524288	1024	4.47

latency for NOISEGEN adds non-trivial area and energy overheads, which we explain next.

4.4.2.1 AESIR-T

We now present the DRAM footprint as well as the area and energy overheads of `AESIR-T`.

DRAM. We first examine the minimum number of noise points that must be stored to meet security requirements. Recall that prior work by Mironov et al. [186] shows that 'gaps' in the random values produced, acts as a side-channel for an attack against DP [186]. Since we use the $bfloat16$ datatype, we must store one of each possible 65,536 ($= 2^{16}$) points to avoid such gaps.

With this minimum number of points in mind, we quantify the increase in memory footprint of DRAM. Table 4.2 shows the footprint of storing different numbers of points in DRAM. Even when storing over half a million points, we add just 1MB of storage. 1MB is less than 5% of the size of the smallest model we evaluate (PreActResNet-18). Thus AESIR only slightly increases the memory footprint of DRAM.

On-chip area and energy. To support AESIR, we add the following hardware to a baseline ML accelerator:

1. A 2 : 1 mux (Figure 4.5)
2. Hardware to scramble values read from DRAM (Figure 4.6) and
3. A hardware RNG for both additions above.

For the RNG, we opt for a cryptographically secure RNG, proposed by Bakiri et al. [16], which provides 9.4 Gbps of randomness. As we run our design at 200MHz, we obtain $9.4 \div 0.2 = 47$ random bits per cycle. For our scrambling hardware, we need 3 (i.e., $log_2(8)$) bits to scramble the 8 muxes we use in our design. We therefore have 44 bits of randomness available per cycle to read random values from DRAM. This allows us to randomly select from 2^{44} addresses in DRAM.

As each location in DRAM contains 8 values, `AESIR-T` supports addressing up to 2^{44+3} (over 10^{13}) random values from memory. However, to address 1MB of stored noise points, we only require $log_2(524288) = 19$ bits from the TRNG. Thus, the TRNG we use supports significant headroom to address far more noise points in memory, if required. In total, `AESIR-T` incurs only a 0.4% area and 0.2% energy overhead, compared to our baseline *DiVa* accelerator, which does not support any noise addition. Of this 0.4% area overheard, 0.319% comes from the RNG hardware while our hardware additions only add 0.082% area overhead.

Table 4.3: Energy overhead of NoiseGen-T

Model	P18	W32	VGG16	Average
Energy overhead (%)	10.05	7.01	7.39	8.15

4.4.2.2 NoiseGen-T

As our training accelerators use 16-bit datatype, we configure NoiseGen to produce 16-bit value every cycle. As shown in Figure 4.7a, our implementation requires a random number generator (RNG) for its operation. Matching prior work [150], our design requires 64 uniform random bits per cycle. However, using the RNG from above, we are only able to obtain 47 random bits per cycle. We therefore include two such RNGs to provide the 64 random bits needed for NoiseGen-T.

On-chip area and energy. Table 4.3 shows the energy overhead of NoiseGen-T for each model that we evaluate. We see that compared to the minimal 0.2% energy overhead imposed by AESIR-T, NoiseGen-T adds an average 8.15% energy overhead. Similarly, NoiseGen-T adds 9.38% area overhead to the baseline accelerator design. This increased overhead is due to: 1) the hardware to generate noise, including our augmentations described in Chapter 4.3 and 2) the extra RNG needed to produce enough random bits. These overheads are significant in the context of power and cost sensitive edge ML accelerators. This demonstrates the benefit of AESIR, compared to adding dedicated noise generation hardware. Note Table 4.3 does not distinguish between CIFAR-10 and CIFAR-100; these models vary only in the number of output classes. This only impacts the size of the last layer which accounts for $< 1\%$ of the total energy in all cases.

4.5 AESIR for robustness

We now evaluate AESIR when used to add noise for adversarial robustness. We first detail our methodology in Chapter 4.5.1. We then then present our evaluation in Chapter 4.5.2. Finally, we detail the oveaheads imposed by AESIR in Chapter 4.5.4.

4.5.1 Methodology

We describe the architecture used to evaluate AESIR for adversarial robustness, followed by our test setup and how we evaluate the adversarial robustness afforded by noise addition.

Architecture description. In this section, we use a new accelerator design – this time targeting inference – based on Eyeriss [36]. Similar to the previous section, NoiseGen-I generates noise in hardware while AESIR-I uses AESIR. Since we target inference, we opt to use an 8-bit fixed-point datatype. As both of these require noise vectors instead of individual noise points, we do not implement our scrambling hardware for inference. Finally, we once again use a DRAM bus-width of 128 bits and operate our designs at 200MHz.

Test setup and parameters. We evaluate the same models as the previous section, shown in Table 4.1. We run all our models using PyTorch 1.11 [212]. We evaluate robustness using the FGSM, PGD and CW attacks, similar to prior work [73, 96]. To carry out our attacks, we use the

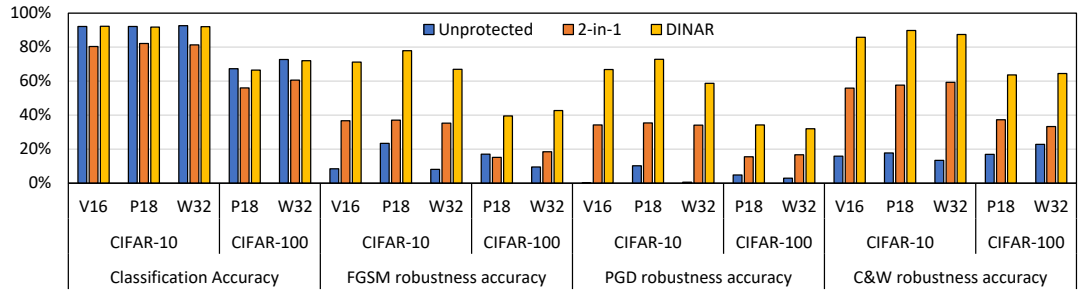


Figure 4.9: Classification and robustness accuracies for evaluated networks. We use the shortened names listed in Table 4.1. (V16 - VGG 16, P18 - PreActResnet-18, W32 - WideResNet32)

TorchAttacks library [141]. We provide all 10,000 test images for each dataset to the library to produce the attacked images. These attacked images are then classified by the model to obtain the robustness accuracy.

We train all models using 32-bit floating point and perform post-training quantization using PyTorch to obtain int8 models. In all cases, we saw a $<1\%$ difference in classification accuracy between the 32-bit floating-point and int8 models.

4.5.1.1 Comparison against prior work

To demonstrate the efficacy of noise addition, we compare against existing hardware techniques for adversarial robustness: 2-in-1 accelerators and defensive approximation. Both of these techniques were described earlier in Chapter 2.3.3.2. We now describe how we run each of the techniques we evaluate.

2-in-1 accelerator : We did not find any code released for this paper, so we instead ran the code from Double-Win Quant [73]. Double-Win Quant is earlier work from the same authors which implements the exact changes outlined in the 2-in-1 accelerator paper. We train models from 4–8 bits and confirm that the code randomly selects from different bit-widths as described by 2-in-1.

Defensive Approximation (DA) : DA is a hardware technique that uses approximate floating point multipliers for increased robustness [96]. DA replaces the full-adders typically used in an array-multiplier with simple buffers. As DA requires custom hardware, we implement the DA approximation using custom CUDA kernels.

WCA : We obtain the WCA code from the author's GitHub repository [64]. WCA includes an additional parameter (M), which is the dimension of the multivariate Gaussian distribution added to the penultimate layer. They recommend that M be greater than the number of output classes to enable high robustness accuracy. We therefore set $M = 32$ for CIFAR-10 and $M = 256$ for CIFAR-100, matching the values used by WCA.

4.5.2 Accuracy

We compare AESIR (implementing WCA) against baseline unprotected models and 2-in-1 accelerator. As DA only applies to floating-point inference, we compare against DA in Chapter 4.5.3 below. Figure 4.9 shows the classification and robustness accuracies for the models we evaluate. We see that the baseline networks achieve high classification accuracy but low

Table 4.4: Comparison of classification accuracy between Defensive Approximation (DA) and AESIR, for 32-bit floating point inference.

Model	Dataset	DA (%)	AESIR (%)
PreActResNet-18	CIFAR-10	10.00	91.75
	CIFAR-100	1.33	66.53
WideResNet-32	CIFAR-10	8.11	92.06
	CIFAR-100	0.18	72.10
VGG-16	CIFAR-10	17.21	92.18
MobileNetV2	CIFAR-10	13.38	92.63

robustness accuracy. This is expected as baseline models have no defences applied, either during training or inference, to improve their robustness.

2-in-1 accelerator. While 2-in-1 improves robustness accuracy compared to the baseline models, it suffers from a drop in classification accuracy. For 2-in-1, we make sure to randomly select a different bit-width for each input, during classification as well as attacks. Despite this, 2-in-1 does not significantly improve adversarial robustness. We believe this is because models trained at various bit-widths do not significantly alter the model's loss landscape. That is, even when switching bit-widths for each input, the different models are similar enough that attacks are still able to successfully find perturbations to fool the model. Thus, the limited randomness of using a fixed number of pre-trained models is insufficient to defend against adversarial attacks.

AESIR. AESIR maintains high classification accuracy, with a $<1\%$ difference compared to the baseline models. AESIR achieves the highest robustness accuracy for all attacks compared to the other techniques. This demonstrates that noise injection is an effective method for improving the robustness of ML models, without sacrificing classification accuracy. We do not show these results for NOISEGEN, as they are identical. Thus, any noise generation technique achieves high robustness and classification accuracy.

4.5.3 Comparison with Defensive Approximation

We now compare AESIR against Defensive Approximation (DA). As DA uses approximate floating-point multipliers, we use floating-point inference. Table 4.4 compares the classification accuracy of models using DA against AESIR. For all models, DA achieves extremely low classification accuracy.

We believe DA sees low accuracy because of batch normalization (BatchNorm) layers in our evaluated networks. BatchNorm layers are widely used in CNNs [124] and allows networks to achieve $\sim$10% higher classification accuracy [239]. DA approximates the multiplier used to compute the floating-point mantissa. Without BatchNorm, repeated multiply-and-accumulate operations cause activations to grow in magnitude through the network. The mantissa in IEEE-754 format only has a range $[1.0, 2.0)$. Thus, as activations grown in magnitude, the error introduced by approximating the mantissa decreases through the network. However, with BatchNorm, activations are normalized through the network keeping them in a small range (e.g., $[-1.0, 1.0]$). This means that the error introduced by DA has a much greater impact on networks that use BatchNorm. As the classification accuracy is so low, we opt not to run any attacks for DA, as we

Table 4.5: Number of possible permutations, space needed and storage overhead (over the smallest model we evaluate) for different amounts of noise points stored in DRAM.

Number of noise values in DRAM	Storage required (KB)	Number of permutations	Overhead (%)
1024	1	2.13599E+96	0.0419%
2048	2	9.174E+105	0.0838%
4096	4	3.9402E+115	0.1676%
8192	8	1.6923E+125	0.3353%
16384	16	7.2684E+134	0.6706%

believe the numbers would not be meaningful.

4.5.4 Overheads

In this section, we detail the overheads of NoiseGen-I and AESIR-I. Once again, we first present the overheads which are common for both, before describing overheads which vary between the two flavours.

Global memory. Similar to DP-ML, we ensure that there is sufficient space in global memory to store random noise vectors when needed. However, since WCA only adds a single noise vector before the penultimate layer, we require far less space in the GLB. Thus, for all the networks we evaluate, we can fit this noise vector in the GLB along with the inputs and weights of the preceding layer without contention.

Latency. Since WCA only requires reading values from DRAM for a single layer in the network, both NoiseGen-I and AESIR-I add minimal latency overhead. For all the networks we evaluate, AESIR-T has a $< 0.01\%$ latency increase, due to performing the noise addition operation. However, doing this for just a single layer adds negligible overhead.

NoiseGen-T must perform a matrix multiplication operation as well as noise addition. This leads to a maximum latency increase of 0.25% for PreActResNet-18, running CIFAR-100. PreActResNet-18 has the fewest layers of all the networks we study. Thus, performing the 256×256 matrix multiplication adds more overhead to this network than the others. Despite both flavours adding only modest latency overheads, we once again see significant differences in area and energy between NoiseGen-I and AESIR-I.

4.5.4.1 AESIR-I overhead

We now detail the overheads imposed by AESIR-I, starting with the DRAM footprint, before moving on to area and energy. However, unlike DP-ML, we do not have a theoretical lower limit for the number of vectors that must be stored in DRAM. Therefore, we opt to perform an empirical analysis to determine this lower limit.

Figure 4.10 shows the impact of varying the number of stored vectors on accuracy. Recall that for WCA, we need vectors of 32 values for CIFAR-10 and 256 values for CIFAR-100. We now show the effect of storing k such vectors on accuracy. For the case where we generate a new point each time (unlimited stored vectors in Figure 4.10), we obtain the baseline accuracy as expected. For the other cases, we pre-compute and store k vectors and for each input, we sample one of

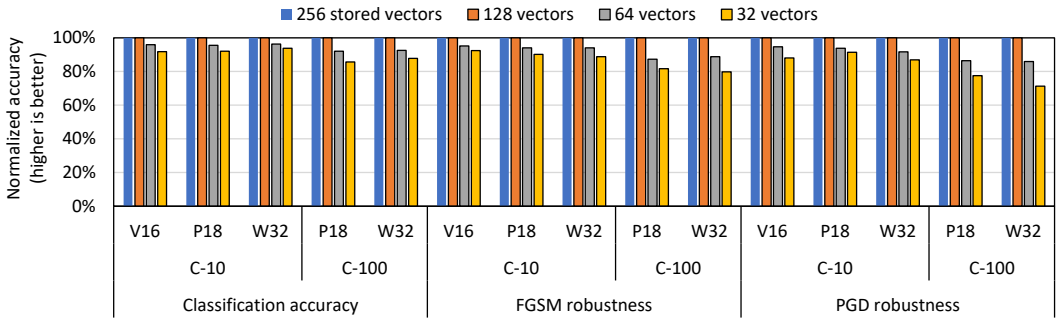


Figure 4.10: Effect of varying the number of stored vectors on classification and robustness accuracies normalized to the baseline accuracy.

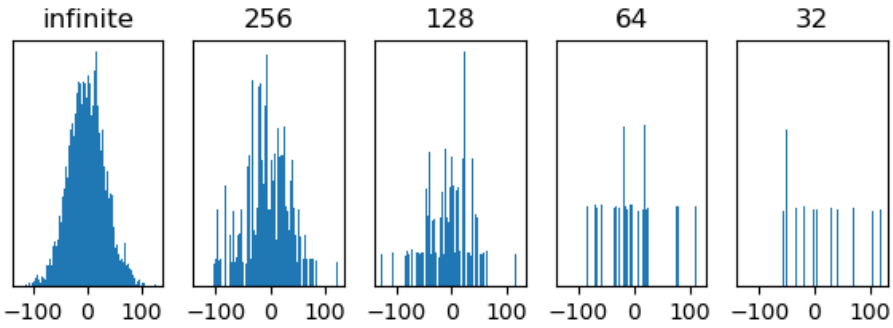


Figure 4.11: Histogram of the Gaussian distributions when sampling varying number of stored points. The title above each plot shows the number of stored points.

these vectors to add to the activations of the penultimate layer. Accuracies are unaffected down to $k = 128$ stored vectors. However, reducing k further results in a drop in both classification and robustness accuracy.

To see why fewer noise points leads to a loss of accuracy, we experiment with reducing the number of points stored from a univariate Gaussian. Figure 4.11 shows histograms generated when varying the numbers of stored points. For each histogram, we compute a list of Q noise points – shown above each figure – and then sample $100,000$ values from this list. We see that for an infinite-sized table, the histogram is unmistakably Gaussian. The x-axis ranges from $[-128, 127]$ as we use an 8-bit datatype.[4] When storing 128 or more values, the histogram maintains a Gaussian shape. But once we go down to 64 or fewer stored values, the distribution starts to appear more uniform. As WCA is trained with a Gaussian distribution, using so few points leads to a loss of accuracy. Therefore, in AESIR we opt to support a minimum of 128 values per dimension.

DRAM footprint. We now analyze the footprint of storing pre-computed noise vectors in DRAM in `AESIR-I`. Table 4.6 shows this footprint for different number of stored vectors. Recall that WCA requires noise vectors of size 32 for CIFAR-10 and 256 for CIFAR-100. Therefore, we show the footprint for both cases, compared to the smallest model we evaluate, PreActResNet-18 (P18). We sweep the number of stored vectors starting from the minimum value of 128, necessary for maintaining accuracy. While AESIR works with far fewer points, even storing 2048 vectors

[4] Despite using 8-bits, setting $Q \leq 256$ does not mean every possible value in $[0, 2^8)$ is stored. As we store a Gaussian distribution, there are many more points close to 0.

Table 4.6: DRAM memory footprint required (compared to the smallest model we evaluate) for varying amounts of noise vectors stored.

Number of noise vectors stored	32 points per vector		256 points per vector	
	Storage (KB)	Footprint (%)	Storage (KB)	Footprint (%)
128	4	0.03	32	0.28
256	8	0.07	64	0.55
512	16	0.14	128	1.10
1024	32	0.28	256	2.21
2048	64	0.56	512	4.42

adds just 0.56% and 4.42% for the 32 and 256 sized noise vectors, respectively. We therefore conclude that AESIR only slightly increases over DRAM memory footprint. For AESIR-I, we require 11 bits from the RNG to access a table of 2048 stored vectors. Using the same RNG as AESIR-T, we have 47 bits from the RNG, providing us with plenty of headroom for storing many more vectors if needed.

On-chip area and energy. AESIR-I adds a 2:1 mux and an RNG, similar to AESIR-T. Together, these only add an additional 0.26% area compared to our baseline accelerator, which cannot add noise. However, as we only require the TRNG before a single layer, we use power gating to disable the TRNG when it is not needed. This results in an energy overhead of $< 0.1\%$ for AESIR-I.

4.5.4.2 NoiseGen-I

To match the 8-bit datatype we use for inference, we use a NOISEGEN configuration which produces 8-bit value every cycle, for NOISEGEN-I. This then reduces the number of uniform random bits required to 32. Thus, a single RNG is also sufficient to provide the required noise for NOISEGEN-I, similar to AESIR-I.

On-chip area and energy. NOISEGEN-T adds 8.05% area overhead to the baseline accelerator. Despite the smaller size of the 8-bit NOISEGEN configuration and the exclusion of extra hardware shown in Figure 4.7b, NOISEGEN-T still adds non-trivial area overhead. This is because of the overall smaller size of the accelerator for int8 vs. bfloat16, which causes even this smaller amount of hardware incur a high area overhead. Similar to AESIR-I, we also employ power gating to reduce leakage power in NOISEGEN-I, resulting in an average energy overhead of just 0.2%.

4.6 Other uses for random noise

We now describe some other ways in which support for random noise can help when running ML algorithms.

Securing split inference. With the increasing use of large ML models on edge devices, prior work has proposed offloading the compute-heavy portions of inference to cloud servers [105, 135, 142, 237]. This requires transmitting partially processed data over a network, which can lead to privacy breaches. Prior works add noise to the data before transmission, to reduce the mutual

information between the original and transmitted data [184, 185, 241]. Titcombe et al. [241] add Laplace noise, while CLOAK [184] adds Gaussian noise. In contrast, SHREDDER [185] adds custom noise distributions – which are learned during training – to the data before transmission. These works target inference using CPUs or GPUs, where the required noise is produced 'on-the-fly'. AESIR enables these algorithms to run on edge accelerators, which do not have CPUs or GPUs.

Variational Autoencoders. In contrast to algorithms that require random noise for security, there are also algorithms that require noise for their basic operation. One such algorithm is Variational Autoencoders (VAE), which are a type of generative model. While VAEs are most commonly used for generating images, they are used for other applications on edge devices as well. For example, VAEs are used for speech enhancement to reduce noise before further audio processing [39].

Weight initialization. A crucial step when training a network is initializing the weights of the network. Setting weights to 0s or uniform random values can prevent training convergence [230]. He et al. [107] propose initializing the network weights with $G^{0,\sigma}$ random values. In this case, σ is determined by the size of the neurons in the preceding layer. For example, to initialize layer L, we set $\sigma^2 = \sqrt{2/size^{L-1}}$. He et al. show that this initialization scheme is essential for convergence in training deep models (i.e., >30 layers).

Regularization. Training neural networks on lead to over-fitting where the networks 'memorize' the training set but fail to generalize to new inputs. One technique to prevent over-fitting is *regularization*, where a small amount of Gaussian noise is added to each input before training [51, 113, 198]. AESIR enables such regularization for models training on edge ML accelerators.

4.7 Related work

We now present related work to our technique. We begin by presenting prior approaches to generating Gaussian noise directly in hardware. We then present efforts which use noise present in analog computations to improve security. Finally, we detail other approaches for securing ML accelerators.

4.7.1 Generating noise in hardware

Efficiently producing noise in hardware has been extensively explored, particularly in the FPGA community. Various techniques have been proposed for generating Gaussian random numbers, in addition to the Box-Muller transform we implement for our comparison. These include the rejection and inversion methods.

Rejection: These techniques employ 'sample-and-reject' techniques, where they produce random numbers and sample them to see if they fit 'inside' the required Gaussian distribution. An example of this is the Ziggurat method, which is often used in software [63, 262]. However, the rejection method is a trial-and-error method, which leads to hardware designs with highly variable latency. As this can significantly complicates the design of hardware, this method has not been favored by hardware designers.

Inversion: This technique uses the inverse of the cumulative distribution function (ICDF) of

the Gaussian distribution to generate random values. However, this ICDF does not have a closed form solution, leading to approaches implementing piecewise linear approximations [62, 101, 151]. Prior work produces Laplace random noise for differential privacy [40]. Using fixed-point can lead to a loss of privacy, due to quantization error; to counter this, the authors propose hardware to post-process the produced noise to maintain privacy. By pre-computing and storing noise points, AESIR allows us to perform any required post-processing ahead of time and avoid the latency cost of doing so at runtime.

4.7.2 Using analog noise

Prior work uses the noise inherent in analog components for robustness during inference. Examples include under-volting the ML accelerator to reduce the transistor error margins and produce random errors as noise [125, 167], and tuning the noise of 6T and 8T SRAM cells, used to store model weights and activations [21]. Roy et al. show that ML accelerators which use crossbars in non-volatile memory to perform calculations can also provide robustness against adversarial attacks [219]. However, achieving robustness using analog noise without sacrificing performance requires careful tuning to account for hardware non-idealities [220]. In contrast, AESIR enables robustness without any hardware fine tuning.

4.7.3 ML accelerator security

There has been many works that aim to make ML accelerators more secure against a variety of attacks. Encryption [117] and Trusted Execution Environments [50] have been proposed to secure cloud-based ML accelerators. GuardNN encrypts all data sent out of the accelerator to ensure security [117]. Dhar et al. [50] add Trusted Execution Environment (TEE) support to ML accelerators, operating in cloud settings. However, the overheads of these approaches would be infeasible on the resource constrained edge accelerators we consider. Other works look at side-channel attacks against ML accelerators such as recovering inputs via power side channels [251] or recovering the network structure by observing off-chip memory access patterns [120]. Inputs can be recovered via a power side channel attack against FPGA-based accelerators [251]. Wei et al. demonstrate a power side channel attack against an FPGA-based accelerator to recover the input [251]. Hua et al. reverse engineer the network structure by observing off-chip memory access patterns in an accelerator that employs dynamic zero-pruning [120]. HuffDuff demonstrates an attack against accelerators that employ sparsity [257]. By observing memory access patterns, they can reduce the search space for possible network structures significantly. All these works rely on side channels, while our work focuses on enabling security-centric ML algorithms on edge ML accelerators.

4.8 Conclusion

We present AESIR, a mechanism to inject noise during network inference to improve robustness against adversarial attacks. We show that AESIR significantly improves robustness accuracy compared to prior hardware techniques, while imposing minimal overheads. Our work is the first to support efficiently adding noise to CNN accelerators. With this work, our goal is to motivate

architects to consider the security implications of their ML accelerator designs. We also describe how adding noise can help ensure the privacy of transmitted user data when network inference is split between the edge and the cloud. AESIR enables architects to explore other areas where added noise can improve the security of ML.

5

EMPATIC: CLUSTERING FOR LOW-PRECISION ADVERSARIAL TRAINING

With the increasing use of ML models, we face a pressing need to defend against anyone who wishes to subvert these models. For adversarial attacks, one technique which has seen extensive use is *adversarial training* [190]. Since it was proposed many years ago, adversarial training remains the most robust technique to date. During adversarial training, we first attack the inputs in the training data before using the attacked images to train the model. Thus, by training the model on the very images that attackers use to fool it, we can improve model robustness. However, as most attacks involve iterative 'steps', adversarial training can significantly slow down training. This is because during each step, we must perform a full pass – forward and backward – through the model.

Many techniques have been proposed to speed up adversarial training, which involve *algorithmic* modifications (as we show in Chapter 2.3.2). In contrast, we leverage a feature of modern GPUs, namely support for high-throughput arithmetic using reduced precision formats. GPUs have emerged as the *de facto* choice for training ML models; their capability to perform very large number of floating-point operations per second makes them an ideal match for the needs of ML training. Traditionally, GPUs were optimized for computations using 32-bit (i.e., single precision) or 64-bit (i.e., double precision) floating point numbers. However, a number of works show that ML models can run at lower precision and still achieve high classification accuracy [194, 195, 269]. Thus, recent GPUs offer higher-throughput using lower precision datatypes, such as 16-bit ($FP16$) and even 8-bit ($FP8$) numbers. As we elaborate in Chapter 5.1.1, some GPUs offer up to 15× higher-throughput for $FP8$ compared to $FP32$.

The popularity of reduced precision formats for inference has also lead to interest in using these types to accelerate training. However, most prior works focus on $FP16$ training [144, 180, 266, 267]. Extending these approaches to $FP8$ is challenging due to the extremely narrow range of this type. During the backward pass of training, we must compute gradients, which can comprise very small values [175, 180]. These small values 'underflow' (i.e., go below the smallest value that can be represented) when training with $FP8$. For instance, going from $FP32$ to $FP16$, the smallest representable value drops from 2^{-149} to just 2^{-24}. This goes down even further to 2^{-9} for $FP8$, making it difficult to use this format 'as-is' for the gradients during

training.

We propose EMPATIC,[1] which retains the throughput advantages but eliminates the challenges of using narrow datatypes. EMPATIC first performs off-line clustering of the training set to identify inputs which are 'close' to each other. Then, during training, we group nearby inputs to be run together. By grouping inputs, the activations through each layer – forward and backward – remain 'close' to each other through the network. We then keep one input from the group as the 'base' and compute the differences (i.e., 'deltas') to the other inputs in the group. This gives us deltas with a smaller range that can fit in $FP8$ datatype. We then compute using these deltas in $FP8$ and $FP16$ for the base values. Finally, we combine both outputs (from the base and the delta computations) to obtain the final layer output. We implement our changes to the widely-used PyTorch framework, including a custom dataloader which supports creating mini-batches from our clustered dataset. We show that EMPATIC effectively leverages hardware support for high-throughput $FP8$ computation in modern GPUs to accelerate adversarial training. While the overall goal of our work is to reduce the time taken for adversarial training, we are unable to show direct speed up results. This is because we do not have access to current GPUs which support $FP8$. Thus, we show that EMPATIC significantly reduces the number of values which underflow the $FP8$ format compared to prior approaches.

Contributions

In this chapter, we make the following contributions:

- We propose EMPATIC, a technique to overcome the challenges of training robust ML models using $FP8$.
- EMPATIC identifies groups of inputs in the training data which are close together. EMPATIC retains one input as a 'base' and computes the difference between the base and the remaining inputs. As we use grouped inputs, these differences are small enough to be represented using the $FP8$ datatype.
- We implement EMPATIC in PyTorch using custom CUDA kernels to minimize overhead.
- We show that EMPATIC significantly reduces the number of values that underflow the $FP8$ format during computations compared to prior approaches. Despite this, EMPATIC is able to achieve comparable classification and robustness accuracy to models trained with $FP16$.

5.1 Background

We now provide some background on the topics required to understand the work in this chapter. We begin with an overview of commonly used floating point formats (Chapter 5.1.1). We then detail existing approaches for training models at reduced precision (Chapter 5.1.2). Finally, we provide an overview of data clustering (Chapter 5.1.3).

5.1.1 Floating point numbers

In this section, we provide an overview of the Floating Point (FP) representation. FP numbers are similar to scientific notation used for writing decimal numbers (e.g., 6.022×10^{-23}). They consist

[1] EMPATIC: Efficient Mixed Precision Adversarial Training through Input Clustering

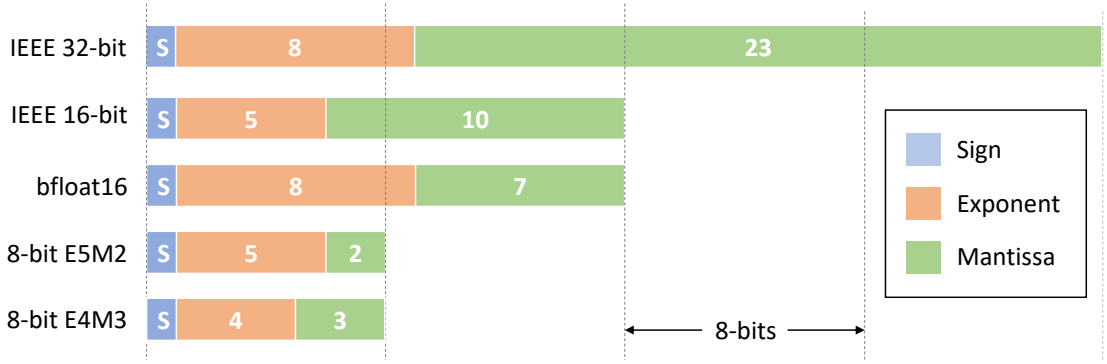


Figure 5.1: The different floating point formats, commonly used for machine learning applications.

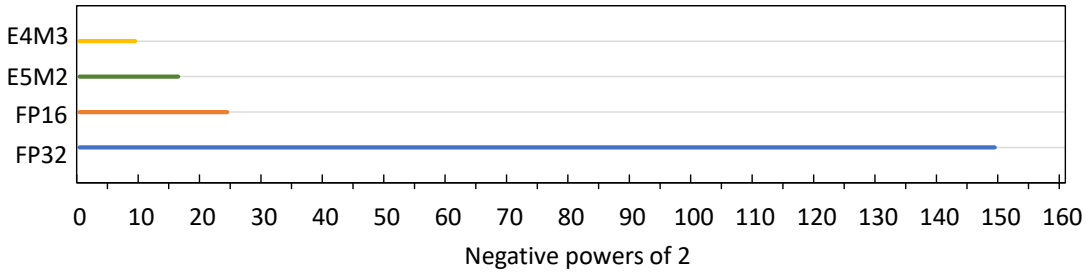


Figure 5.2: Range of values representable in different floating point formats. The x-axis shows the range is negative powers of 2.

of three parts: the sign bit (s), the exponent (e) and the mantissa (m). To cover both positive and negative exponent values, the exponent is stored in 'biased' format. Thus, the number can be written numerically as: $(-1)^s \times m \times a^{e-b}$, where a is the base (typically 2 in digital systems) and b is the bias exponent.

Common formats: We now describe FP number formats commonly used in ML and depict them in Figure 5.1. The industry standard for FP numbers is IEEE-754 and includes specifications for 16-, 32-, 64- and 128-bit formats [123]. We focus on 16- and 32-bit cases as the 64- and 128-bit formats are rarely used for ML. We refer to these as $FP16$ and $FP32$ throughout this chapter. Recently, GPUs from both Nvidia and AMD have included support for $FP8$ formats, including *E5M2* and *E4M3* [5, 181]. Recent GPUs provide a significant increase in throughput with reduced precision types. The FP hardware in these GPUs can perform multiple narrow-precision FP operations in the same time as a single wider-precision operation.

Figure 5.3 shows the speed-ups when moving to reduced precision formats, for the last 5 generations of Nvidia GPUs. We gather this data from the datasheet for each GPU generation [201, 202, 204, 238]. Normalized to the $FP32$ throughput, we see that there has been an increasing trend to speed up $FP16$ and $FP8$ operations. Bars shown with 'TC' indicate support using specialized 'Tensor cores' – optimized for ML – as opposed to the regular 'SIMD cores', which are used for other GPU compute operations. Indeed, we see that $FP16$ is the widest precision supported on Tensor cores, which highlights the trend of using narrow types for ML workloads. When using Tensor cores, we see that performance also varies drastically based on the datatype that values are accumulated into. For example, for the Turing generation, accumulating $FP16$

Table 5.1: Comparison of commonly used floating point formats. We show the largest and smallest values which can be represented in each format as well as the abbreviation we use for each.

Format	Abbreviation	Smallest representable value	Largest representable value
IEEE 32-bit	FP32	1.401×10^{-45}	3.34×10^{38}
IEEE 16-bit	FP16	5.96×10^{-8}	65504
8-bit E5M2	E5M2	1.525×10^{-5}	57344
8-bit E4M3	E4M3	0.00195	448

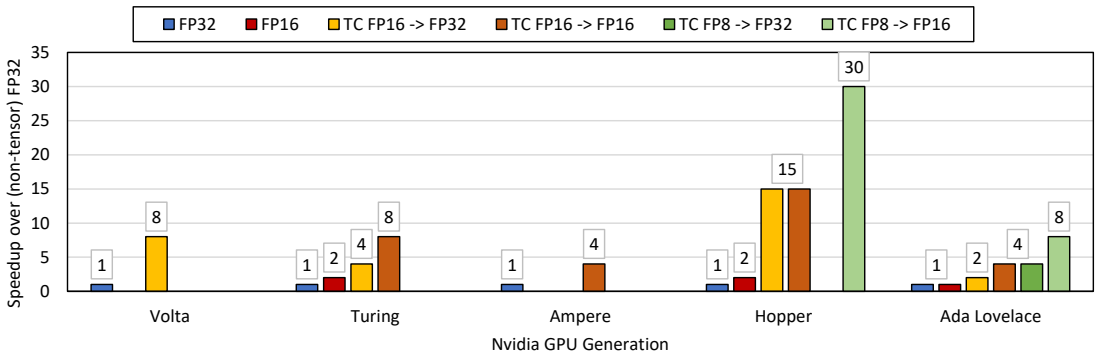


Figure 5.3: Performance comparison for different floating point formats across generations of NVidia GPUs. Bars with 'TC' show performance when using Tensor cores. 'a→b' indicates values of FP type 'a' accumulated into type 'b'.

values into $FP16$ is $2\times$ faster than accumulating into $FP32$. While the speed-ups when going from $FP32 \rightarrow FP16 \rightarrow FP8$ have varied from generation to generation, for the latest 'Ada Lovelace' generation, we see a more even trend of $2\times$ performance increase when moving from a larger to a smaller format. In general however, we see that performance increases substantially when using narrower FP types.

Trade-offs: Despite the increase in throughput when using $FP8$, it is not always possible to simply replace $FP32$ or $FP16$ operations with $FP8$. In the next section, we elaborate on the challenges with doing so for ML training. This is because of the drastic reduction in the range of values which can be represented as we reduce the datatype size. We list the range for the formats in Figure 5.1 in Table 5.1. To highlight the scale of the range reduction, we also depict this visually in Figure 5.2, using a log-scale. The range of values that can be represented decreases drastically, going from $FP32$ to the smaller formats. We see that even the much smaller range of $FP16$ is further reduced in both $FP8$ types. Thus, effectively using $FP8$ for computations which produce values across a wide range (such as during training) is challenging. We elaborate further on this specific scenario next.

5.1.2 Reduced precision training using loss scaling

Using reduced precision formats to speed-up inference has been widely studied in previous works. Prior work has shown that trained networks can be quantized to operate with just 4 or 8 bits per value, with minimal accuracy loss [194, 195, 269]. Binarized neural networks have even shown that inference can be run with just 1 bit per value [121, 217]. The majority of works targeting

Algorithm 4: Stochastic Gradient Descent (SGD), incorporating loss scaling.

1 **Inputs:** Examples ($x_1 \ldots x_N$), total timesteps (T), weights (W), minibatch size (B), loss function ($\mathscr{L}$), learning rate (η), loss scale (ζ)
2 **Initialize** model weights before timestep 0: W_0
3 **for** $t \in [T]$ **do**
4 Randomly sample NB minibatches.
5 **for** $b \in [NB]$ **do**
  ▷ Compute the scaled loss.
6 $\mathscr{S} \leftarrow \zeta \times \mathscr{L}(W_t(x_b, y_b))$
  ▷ Compute gradients
7 $g_t(x_b) \leftarrow \nabla_{W_t} \mathscr{L}(W_t(x_b, y_b))$
8 **end**
  ▷ Sum gradients from all minibatches
9 $\widetilde{g}_t = \frac{1}{B}\left(\sum^{NB} g_t(x_b)\right)$
  ▷ Update weights for next time step
10 $W_{t+1} \leftarrow W_t - \frac{\eta \widetilde{g}_t}{\tau}$
11 **end**

inference use 'fixed point' rather than the floating point format we described in the previous section. However, for the same number of bits, fixed point formats have even narrower range that floating point, making them even harder to use for training.

Given the interest in using reduced precision types for inference, there has also been many works that look at using these types for training. During training, we first perform the forward pass through the network to compute the loss function. We then compute the gradients and back-propogate these through all the layers (as we showed in Algorithm 1). However, these gradients can be very small values and 'underflow' (i.e., go below the smallest representable value) when using reduced precision formats [175, 180]. When this happens, very small values are simply set to 0 and the information in those gradients is lost. Despite this, there is a pressing need to effectively use these narrow formats during training, as GPUs continue to offer higher throughput for these formats.

Loss scaling: A widely used technique to prevent gradient underflow during training is 'loss scaling'. For loss scaling, we first multiply the loss values with a pre-determined scaling factor [144, 180]. Loss scaling 'shifts' the gradients to make more effective use of the entire range of reduced precision datatypes.

Algorithm 4 shows the steps for performing loss scaling during training. We multiply the loss by ζ (Line 6) before computing the gradients (Line 7). Finally, when performing the 'weight update' step, we must first divide the gradients by ζ (Line 10). This compensates for the multiplication with ζ done earlier. While ζ can be any large value, it is typically chosen to be a power of 2. As FP numbers use a base of 2, multiplication and division by ζ only requires changing the exponent (e); this can be done exactly with no rounding error.

Drawbacks of loss scaling: While loss scaling has been shown to be effective for many networks, it suffers from two drawbacks. First, the scale factor must be set before network training. This involves a 'trial-and-error' process which can be time consuming. We typically start with a large ζ value and check the outputs of each layer after Line 6 in Algorithm 4. If any

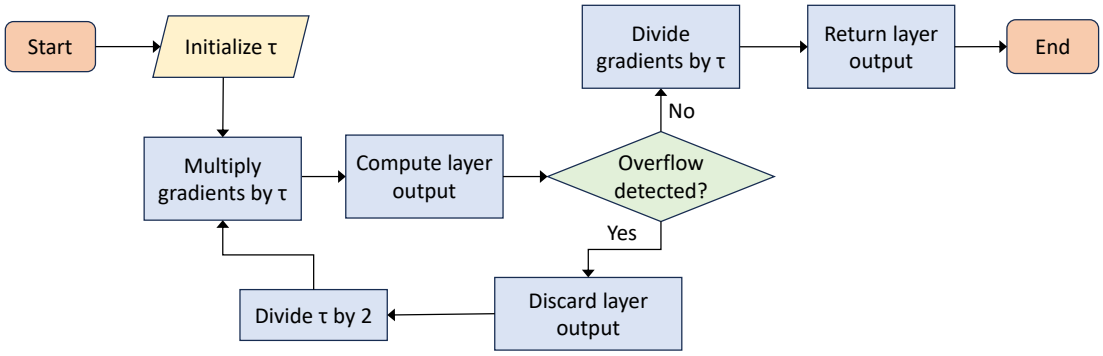


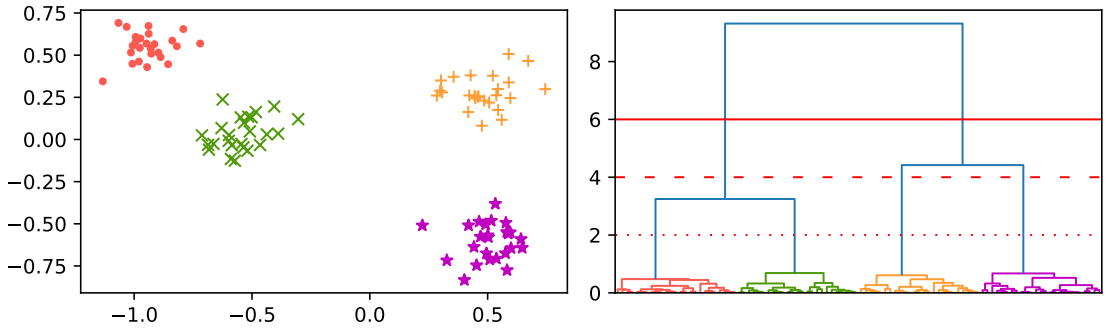
Figure 5.4: Steps taken per layer for adaptive loss scaling.

of the results 'overflowed' (i.e., exceeded the maximum possible value for the given datatype), we must reduce ζ and re-train. Thus, determining the optimal scale factor can require many training runs, which can extend the overall training time significantly. Also, loss scaling uses a single value to scale the gradients for every layer. This can lead to sub-optimal accuracy because the range of gradients can vary across layers. Thus, a ζ value chosen to avoid overflow in one layer may not fit the ideal range of another layer and still lose information due to underflow.

Adaptive loss scaling: Adaptive loss scaling has been proposed to overcome the challenges of fixed loss scaling for $FP16$ training [266, 267]. This technique uses a *per-layer* scale factor to allow gradients in each layer to best use the full range of reduced precision formats. We denote these per-layer scale factors as τ to differentiate them from the earlier network-wide scale factor of ζ. Figure 5.4 shows the steps to employ adapative loss scaling. We describe the process for a single layer and note that the same process applies to all network layers. First, we initialize the per-layer scale factors to be a power of 2. We then scale the gradients by τ before computing the layer output. Now, we check for overflow in the output; if overflow is detected, the computation is discarded and τ is halved. We then recompute the layer output using the new τ. This adaptive process allows each layer to adjust to a different scale factor. However, as we show in Chapter 5.4.2, adaptive loss still does not allow training using $FP8$. Despite this, we use adapative loss scaling in EMPATIC (with some modifications), as we explain in Chapter 5.2.1. We now provide some background on data clustering, which is the basis for our technique to allow training using $FP8$.

5.1.3 Data clustering

Clustering is an unsupervised learning technique that is used to glean insights from large datasets. Clustering is most commonly used to identify groups of datapoints which are close together to form 'clusters'. In this section, we provide some background on techniques commonly used for clustering. The ML datasets that we cluster have very high dimensionality, making them a challenge to cluster. Therefore, we also describe techniques to reduce the dimensionality of our data before clustering. Finally, we also describe metrics which are used to evaluate the quality of clustering techniques.



(a) Random dataset with 4 clusters.

(b) Dendogram showing Hierarchical Agglomerative Clustering.

Figure 5.5: Example of clustering: (a) a random dataset of 4 clusters and (b) the dendogram, after performing hierarchical agglomerative clustering.

5.1.3.1 Clustering techniques

Clustering techniques can be broadly split into two categories: centroid-based and connectivity-based.

Centroid-based: The key idea with centroid-based clustering is to treat the center of a set of datapoints as the cluster center. K-Means clustering is the most famous example of centroid-based clustering. K-Means partitions data into k clusters and assigns points to each cluster by minimizing the mean distance to the cluster centroid. In general, centroid-based clustering is NP-hard problem. Thus, numerous heuristic solutions exist to speed this up. PAM is a optimization which uses greedy search to speed up the process [137]. CLARA is a further refinement to PAM which uses sampling (instead of using all the datapoints) to speed up clustering [138].

Despite their popularity, centroid-based clustering techniques suffer from two drawbacks, which makes them unsuitable for our purposes. 1) We need to know the number of clusters ahead of time for centroid-based clustering. We do not have a preset number for this when clustering our datasets. 3) Centroid-based clustering is designed for data that is in 'discrete' clusters. We know that ML datasets are not in such discrete clusters [270]. Indeed, if this were the case, these datasets could easily be classified using simple linear classifiers. For these two reasons, we do not use centroid-based clustering for our technique.

Connectivity-based: In contrast to centroid-based techniques, connectivity-based techniques aim to construct a hierarchical relationship between the data for clustering. They begin by considering each datapoint as a separate cluster and then merge clusters based on the distance between them. The simplest form of connectivity-based clustering is Hierarchical Agglomerative Clustering (HAC) [191]. Using HAC, we combine clusters iteratively based on the distance between them, a process that is best depicted visually.

Figure 5.5 shows an example of clustering a random dataset with 4 clusters. In Figure 5.5(a), the points belonging to each cluster are indicated with different colors and symbols for easy identification. Figure 5.5(b) shows the dendogram (or 'tree diagram') as we perform HAC on this dataset. To match Figure 5.5(a), the clusters are colored are the same in Figure 5.5(b). The y-axis in Figure 5.5(b) shows the intra-cluster distance. Each vertical line corresponds to a single cluster, starting from 2 clusters at the very top to 100 at the bottom. As we vary the distance for

clustering, we get different numbers of clusters. For example, for a distance of 6 (solid red line in Figure 5.5(b)), we get just 2 clusters: 1) red and green and 2) orange and purple. As we go to a distance of 4 (dashed red line), we increase to 3 clusters; orange and purple are now in separate clusters. Finally, at a distance of 2 (dotted red line), we get the 4 clusters we expect. Of course, if we reduce the distance even further, we would start to separate this into even smaller clusters, until we eventually have each point being in its own cluster. Thus, it is important to know where to 'cut' the dendogram to get the appropriate number of clusters. We do this in our work using a metric which we describe in Chapter 5.1.3.3.

Linkage method: HAC works by grouping clusters based on their intra-cluster distance. Measuring the distance between single points is straightforward. However, as we group points into clusters, there are multiple methods to measure the distance between clusters.

We evaluate three such methods, which have been shown to be effective for high-dimensional data: median, centroid and ward [191]. **Median** computes the centroid of the new cluster as the average of the centroids of the original two clusters. **Centroid** recomputes the centroid for the new cluster based on all the points in the original two clusters. **Ward** uses the Ward variance minimization algorithm [250]. The Ward algorithm combines clusters such that the sum of squared inter-cluster distance is the least increased.

Like other clustering techniques, HAC also struggles to scale to the large numbers of dimensions used in real-world datasets. Therefore, we require a way to reduce the dimensionality of our data before clustering, which we detail next.

5.1.3.2 Dimensionality reduction

We now describe approaches to reduce the dimensionality of data, which are commonly used to improve the outcome of clustering.

Principal Component Analysis: A widely-used technique for dimensionality reduction is Principal Component Analysis (PCA) [93]. Given a dataset of size $(n \times n)$, PCA works in four steps:

1. Compute the covariance matrix of the normalized dataset.
2. Compute the eigenvectors and eigenvalues of the covariance matrix.
3. Sort the eigenvalues in descending order and choose the top-k eigenvalues to form the 'principal' components.
4. Project the original data onto the principal components to create a lower-dimensional representation.

Selecting the appropriate k value is essential for ensuring the quality of PCA output. A common technique to determine k is to use the Explained Variance Ratio (EVR) [132]. EVR is the cumulative percentage of the the total variance that is explained by each additional principal component. We want k to be large enough to represent a high percentage (over 90%) of the variance in the dataset. This tells us that our reduced dataset is still representative of the original dataset, without losing too much information.

Neural Network Embedding: Another approach to dimensionality reduction is to use neural networks to create 'embeddings', which are low-dimensional representations of high-dimensional

data [270]. Hinton et al. were the first to do so by using an autoencoder [112]. Autoencoders consist of two networks: an encoder ($z = f(x)$) and a decoder ($x' = g(z)$). The autoencoder is then trained to reduce the error between the original input (x) and the reconstructed input x'. For dimensionality reduction, the size of the encoder output z is made much smaller than x. We then discard the decoder and simply use the encoder output as our lower-dimensional projection of the input data. Many prior works have explored using an autoencoder, followed by an traditional clustering technique (e.g., HAC) to cluster machine learning datasets [95, 172, 196]. These works show that using encoders achieves higher clustering accuracy compared to traditional techniques such as PCA.

5.1.3.3 Evaluating clustering techniques

Finally, we describe how we evaluate the quality of a given clustering technique. Metrics which are used to measure classification accuracy (i.e., accuracy, F-score, area under the curve) are also used to evaluate clustering [95, 172, 196]. Prior works use these metrics with the class labels to measure accuracy. Thus, with these metrics, methods which group the most inputs from the same class together are deemed to be more accurate. However, this approach is unsuitable for us as we do not have a final desired grouping in mind. Thus, we cannot use a metric which requires *a priori* class labels.

We instead use a metric which does not require labels, namely silhouette scores [218]. The silhouette score measures how similar a point is to its own cluster compared to other clusters (separation). The score ranges from -1 to $+1$, which a higher value indicates that each point is well matched to its own cluster. We then report the average score for all points in the dataset. With the background covered, we now describe EMPATIC, our technique for using $FP8$ hardware for robust ML training.

5.2 EMPATIC

In this section, we describe EMPATIC, our approach to speed up adversarial training. We begin with a high level overview (Chapter 5.2.1). We then describe the modified dataloader we implement to support clustered data (Chapter 5.2.2). Finally, we explain the changes we make to layer implementations to support EMPATIC (Chapter 5.2.3).

5.2.1 High level overview

With EMPATIC, our goal is to speed up adversarial training by using the capability of modern GPUs for high-throughput $FP8$ computations, without suffering the drawbacks of prior work. Specifically, we want to reduce the range of values during computation, so that they can fit into the $FP8$ format, without underflow. In particular, we want to leverage the ability of modern GPUs to perform high-throughput $FP8$ multiplication operations. We design our technique to speed-up adversarial training. Specifically, we target the iterative steps during attacks used to compute the perturbed inputs. During attacks, we only calculate the gradient of the inputs with respect to the weights. It is important to note that our approach only applies to computing the gradients of the inputs; the derivation below does not apply for weight updates.

We now provide the mathematical formulation of our approach. We write matrices with uppercase letters and vectors using lowercase letters. Consider a single matrix multiplication of a set of input vectors $\mathbf{X} \in \mathbb{R}^{n \times k}$ by a weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ to yield an output vector $\mathbf{y} \in \mathbb{R}^{m \times k}$. This can be the forward or backward pass of a convolutional or fully connected layer, as they are all computed using matrix multiplication. We calculate the output $\mathbf{y}$ as:

$$\mathbf{Y} = \mathbf{W}\mathbf{X} \tag{5.1}$$

Rewriting $\mathbf{W}$ and $\mathbf{X}$ in Equation 5.1 to separate just the first column and row in each, we get:

$$\mathbf{Y} = \begin{bmatrix} \mathbf{w_0} & \mathbf{W'} \end{bmatrix} \begin{bmatrix} \mathbf{x_0} \\ \mathbf{X'} \end{bmatrix} \tag{5.2}$$

In Equation 5.2, $x_0 \in \mathbb{R}^{1 \times k}$, $\mathbf{w_0} \in \mathbb{R}^{m \times 1}$, $\mathbf{W'} \in \mathbb{R}^{m \times (n-1)}$ and $\mathbf{X'} \in \mathbb{R}^{(n-1) \times 1}$. Multiplying these matrices we obtain:

$$\mathbf{Y} = \mathbf{w_0}\mathbf{x_0} + \mathbf{W'}\mathbf{X'} \tag{5.3}$$

To implement EMPATIC, we subtract x_0 from $\mathbf{x'}$ to get:

$$\mathbf{Y} = \underbrace{\mathbf{w_0}\mathbf{x_0}}_{①} + \underbrace{\mathbf{W'}(\mathbf{X'} - \mathbf{x_0}\mathbf{1_{n-1}})}_{②} + \underbrace{\mathbf{x_0}\mathbf{W'}}_{③} \tag{5.4}$$

In Equation 5.4, $\mathbf{1_{n-1}} = (1, 1, ...1) \in \mathbb{R}^{(n-1) \times 1}$. For ease of referencing, we label the three terms in Equation 5.4 sequentially as ①, ② and ③. With our modifications, we are subtracting extra terms which must them be added back in to preserve the equality. We do this by adding the ③ in Equation 5.3. Next, we analyze the increase in the number of operations due to our modifications and the expected impact of this on runtime.

5.2.1.1 Computational complexity

The goal of our modifications is to obtain 'deltas' (② in Equation 5.4). By picking the appropriate input values for $\mathbf{X}$, we can get deltas with a smaller range than the original values. ① are computations using unmodified 'base' values (i.e., $\mathbf{x_0}$) which are performed as normal. As we know that regular computations cannot be performed using $FP8$, we compute both ① and ③ using $FP16$ and only use $FP8$ for computing ②. We also keep all our values in $FP16$ in memory and only convert them to $FP8$ before using them (as explained in Chapter 5.2.3 below).

To quantify the computational complexity, consider the scenario where both $\mathbf{X}$ and $\mathbf{W}$ are both $n \times n$ matrices. In this case, ① and ③ are both matrix-vector products and have $\mathcal{O}(n^2)$ complexity. ② is a matrix-matrix product and has $\mathcal{O}(n^3)$ complexity. Thus, by implementing ② using $FP8$, we speed-up the most computationally complex operation in Equation 5.4. As n increases, the complexity of ② grows faster than ① and ③ and we expect to get closer to the time to perform the entire operation in $FP8$. Indeed, we experimentally validate this observation in Chapter 5.4.3. With the mathematical overview of our approach completed, we now discuss some challenges with the implementation of our technique.

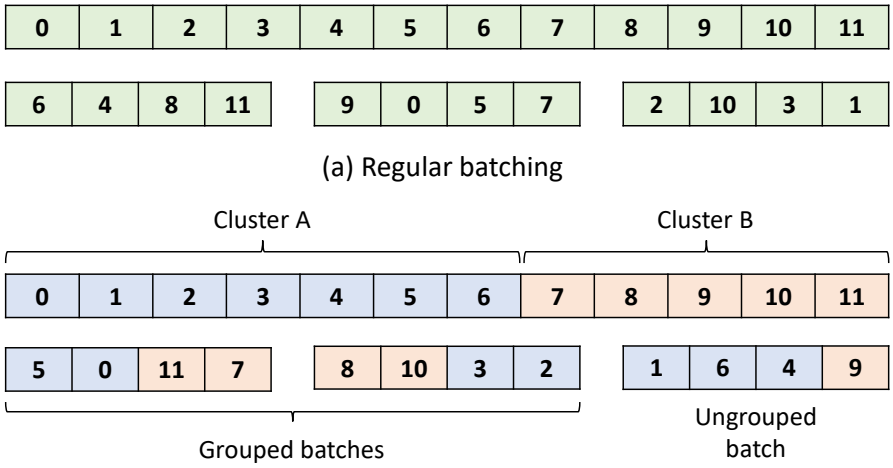


Figure 5.6: Regular vs grouped batching.

5.2.1.2 Implementation challenges

We now discuss two issues we must overcome when implementing EMPATIC and how we address these.

Preventing underflow: As we perform clustering for the gradients, we are dealing with very small values which typically underflow $FP8$. When we then perform a subtract operation between two such values, the result becomes even smaller and we risk even more underflows. To prevent this, we use adaptive loss scaling which we described in Chapter 5.1.2. Previous adaptive scaling techniques use a single scale factor (τ) per layer. In our case, we opt to use two scale factors: One for the 'base' values and another for the deltas. As we use $FP16$ for our base values, adapative loss scaling works well for these values. Then, by scaling the deltas rather than the inputs, we also overcome the issues with adapative loss scaling $FP8$.

Reducing the range of deltas: If we simply picked random points from the training set, the deltas are not likely to be smaller than the base values. To ensure the deltas are as small as possible, we must first cluster the training inputs. Then, by picking groups from training inputs which are closest to each other, we obtain the small deltas we need. Furthermore, as similar inputs excite neurons in a similar way, the deltas remain small through all the layers in the network, in both the forward and backward passes. Key to our approach is clustering the training inputs effectively to obtain small deltas.

5.2.2 Supporting clustered data

With EMPATIC, we cluster the entire training set into C clusters. Then, during each epoch, we must sample inputs from these clusters to form the batches we require for training. As this is not normally done during training, we implement the functionality required to support this. We implement all our modifications to the PyTorch framework [212]. In PyTorch, loading a dataset and forming batches is performed by the 'dataloader' class. Typically, for a batch size b and N inputs, we obtain m batches, where $m = \lceil N \div b \rceil$. We show this in Figure 5.6(a), where $N = 12$ and $b = 4$. During training, it is common to randomly shuffle the inputs to improve accuracy.

Listing 5.1: PyTorch code for Conv2d

```
class conv2d:

  def forward(context, inputs, weights, bias, conv2d_params):
        context.save_for_backward(inputs, weights, bias,
           ↪ conv2d_params)
        outputs = conv2d_forward(inputs, weight, bias, parameters)
        return outputs

  def backward(context, grad_output):
        inputs, weights, bias, conv2d_params = context.saved_tensors
        grad_input = grad_weight = grad_bias = None

        if context.compute_input_gradients:
          grad_inputs = conv2d_input_grad(inputs, weights,
             ↪ grad_output, conv2d_params)

        if context.compute_weight_gradients:
          grad_weights = conv2d_weight_grad(inputs, weights,
             ↪ grad_output, conv2d_params)

        if context.compute_bias_gradients:
          grad_bias = conv2d_bias_grads(bias)

        return grad_inputs, grad_weights, grad_bias
```

However, we cannot directly use the default dataloader for clustered inputs. If we create shuffled batches, inputs from each cluster may no longer be contiguous. In this case, our deltas would no longer fit within a small range and we are likely to exceed the range of $FP8$.

To enable batching with clustered inputs, we develop a custom dataloader. Our dataloader also takes into account the number of inputs which must be kept contiguous. This value (k) controls the number of 'delta' inputs computed; there is always 1 base value and $k-1$ delta values. k is independent of the batch size b. Thus, each batch can have multiple 'groups' of inputs, each from different clusters. Figure 5.6(b) shows grouped batching for a dataset with 2 clusters and $k = 2$.

Our approach randomly picks groups to fill every possible batch of inputs. For all these batches, we have k inputs from each cluster placed together. However, there may be some inputs which cannot be grouped. To accommodate this, we collect all the ungrouped inputs into a single batch. We then run this final batch as usual, without any clustering.

Our approach does not limit the choice of dataset or the batchsize. Furthermore, our approach still implements 'full shuffling' of the dataset; every input in the dataset is still used exactly once in each epoch. We do this as prior work shows that shuffling is important to avoid over-fitting and allow for faster training convergence [155, 178]. Our approach also picks k random inputs from each cluster to form groups each time. This ensures that the same k inputs are not always run contiguously in every epoch. Finally, we also support clusters with fewer than k inputs. We simply consider such inputs to be 'ungrouped' and always add them to the final batch.

5.2.3 Layer modifications

We now describe how we modify layers to support clustered inputs. Similar to the modified dataloader, we also implement our layer modifications in PyTorch. While EMPATIC can be implemented for a wide variety of layer types, we choose to only implement EMPATIC for 2D convolutional (i.e., 'Conv2d') layers, as they account for the majority of the runtime of CNNs [156]. To understand the changes we make, we first describe the baseline implementation of Conv2d in PyTorch.

Conv2d in PyTorch: Listing 5.1 shows the PyTorch code which implements the baseline Conv2d layer. We begin with the forward pass (Line 3). First, we save the layer parameters so we can use them during the backward pass, using PyTorch's *context* functionality (Line 4). For clarity, we group the different parameters required for Conv2d (e.g, kernel sizes, stride, padding etc) as *conv2d_params*. We then calculate the output using the *conv2d_forward* function provided by PyTorch (Line 5). This function transforms the input and weight to two large matrices, which are then multiplied to produce the output.

The backward pass (Line 8) is more complicated as we can compute the gradients for three different tensors: the inputs, the weights and the bias. The *context* also stores the information which tells us which tensors we must compute the gradients for. Input gradients (Line 12) are computed during attacks and when we update the model weights. Weight gradients (Line 15) are only computed during weight updates. Finally, bias gradients (Line 18) are computed when the layer has bias values.

5.2.3.1 EMPATIC modifications

We modify two of the functions shown in Listing 5.1. Specifically, we implement custom versions of the *conv2d_forward* (Line 5) and *conv2d_input_grad* (Line 5) functions. Our modified functions are 'drop-in' replacement for the baseline PyTorch versions and require no other changes. We now describe the changes we make to each of these functions.

Forward: Many prior works show that the forward pass can be performed using $FP8$ without information loss [194, 195, 269]. We want to take advantage of this and perform the forward pass in $FP8$. However, converting all the weights and activations to $FP8$ using PyTorch functions would be very expensive, both in terms of latency and memory. This is because PyTorch only supports changing the underlying datatype by creating a copy of the data. Instead, we modify the underlying CUDA implementation of the $Conv2D\_forward$ function to cast $FP16$ values to $FP8$ just before they are used. We do this using a special $FP8$ CUDA instruction that takes just one cycle. We then perform the same calculations as $Conv2D\_forward$ using $FP8$ and return the result as $FP16$. This is because we want to maintain all the values as $FP16$ and only use $FP8$ to accelerate computations. As Nvidia GPUs natively support accumulating $FP8$ into $FP16$, this final step adds no additional overhead.

Backward: In the backward pass, we only modify the *conv2d_input_grad* function (Line 5). We do so as our goal is to speed-up adversarial training, where carrying out the attack takes the majority of the time. During attacks, only the input gradients are computed, as the model itself is not changed and so we do not compute weight gradients.

Algorithm 5: Backward pass with clustering.

1 **Inputs:** Batch of output gradients (O), weights (W), Conv. params (P), cluster size (k), base scale factor (τ_{base}), delta scale factor (τ_{deltas})
2 **Outputs:** Batch of input gradients (I)
▷ `Split the output gradients into the base and deltas`
3 $O_{base}, O_{deltas} \leftarrow Split(O, k)$
▷ `Scale the base and deltas using the per-layer scale factors`
4 $O'_{base} \leftarrow O_{base} \cdot \tau_{base}$, $O'_{deltas} \leftarrow O_{deltas} \cdot \tau_{deltas}$
▷ `Compute the inputs gradients for the base values`
5 $I'_{base} \leftarrow conv2d\_input\_grad(O'_{base}, P)$
▷ `Compute the inputs gradients for the deltas values`
6 $I'_{deltas} \leftarrow conv2d\_input\_grad\_custom(O'_{deltas}, P)$
▷ `Scale the base and deltas using the per-layer scale factors`
7 $I'_{base} \leftarrow O'_{base} \div \tau_{base}$, $I'_{deltas} \leftarrow O'_{deltas} \div \tau_{deltas}$
▷ `Combine the base and delta input gradients`
8 $I \leftarrow Combine(I_{base}, I_{deltas})$
9 **if** $CheckOverflow(O'_{base})$ **then**
10 $\quad \tau_{base} \leftarrow \tau_{base} \div 2$
11 **if** $CheckOverflow(O'_{deltas})$ **then**
12 $\quad \tau_{deltas} \leftarrow \tau_{deltas} \div 2$

Algorithm 5 describes our implementation of the *conv2d_input_grad* function. For clarity, we show this using pseudo code instead of PyTorch code. As inputs, we have the gradients of the outputs to this layer (O), the layer weights (W) and the parameters for this layer (P) – such as the kernel sizes, stride, padding etc. As mentioned above, O and W are both stored using $FP16$. As we explained in Chapter 5.2.1, we use two scale factors per layer. In total, our modifications require: the cluster size (k) and the scale factors for the base and delta values (τ_{base} and τ_{deltas}). We now explain the function of each line of Algorithm 5.

We begin by splitting the output gradients O into the base (O_{base}) and deltas (O_{deltas}) (Line 3). We then compute the output gradients O'_{base} and O'_{deltas} (Line 4). We compute the (scaled) input gradients for the base values I_{base} using $FP16$ (Line 5). For this we use the regular PyTorch function to compute the input gradients (*conv2d_input_grad*). However, to compute the (scaled) delta input gradients I_{deltas} using $FP8$, we implement a modified version of *conv2d_input_grad*, namely *conv2d_input_grad_custom* (Line 6). Similar to the modified forward pass, *conv2d_input_grad_custom* casts the $FP16$ values to $FP8$ for the computation and returns $FP16$ results. As I'_{base} and I'_{deltas} are both scaled, we must first 'unscale' them (Line 7). Finally, we combine these to obtain the final input gradient I.

Adaptive loss scaling: We implement adaptive loss scaling in EMPATIC. We choose an initial value of $32,768$ for both τ values, which is the largest power of 2 that can be represented in $FP8$. If an overflow is detected, we adjust the appropriate τ value by dividing it by 2 (Lines 10 and 12). We want τ to be a power of 2 so that multiplication and division can be performed precisely in floating point, without any rounding error. Thus, dividing τ by 2 allows us to maintain this feature. With our technique described, we now move onto presenting the results of our clustering

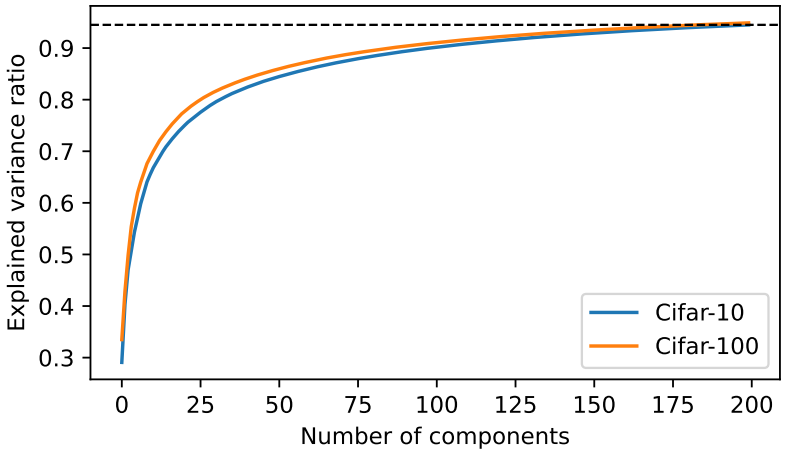


Figure 5.7: PCA explained variance ratio vs. number of components for CIFAR-10 and CIFAR-100 datasets.

experiments.

5.3 Clustering

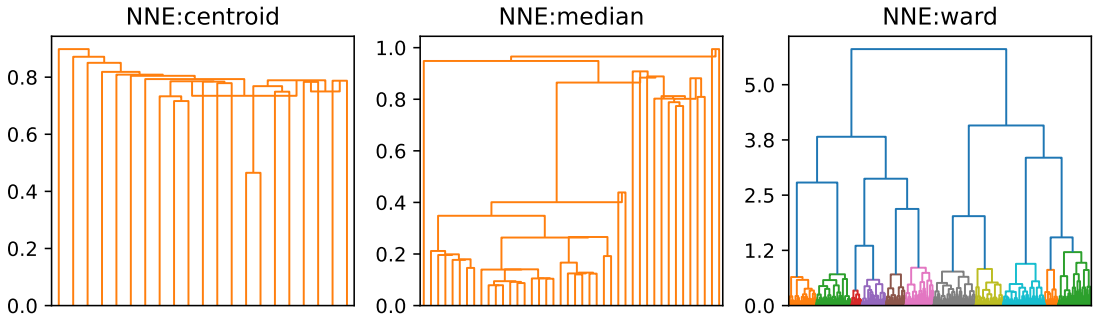
In this section, we describe how we perform clustering on training data. Directly clustering the datasets typically used in ML is intractable, due to the huge number of dimensions. For example, the CIFAR-10 and CIFAR-100 datasets we evaluate have $3 \times 32 \times 32 = 3072$ dimensions each. We therefore use techniques to first reduce the dimensionality of the datasets before clustering. To perform our experiments, we use the SciPy framework [244].

5.3.1 Dimensionality reduction

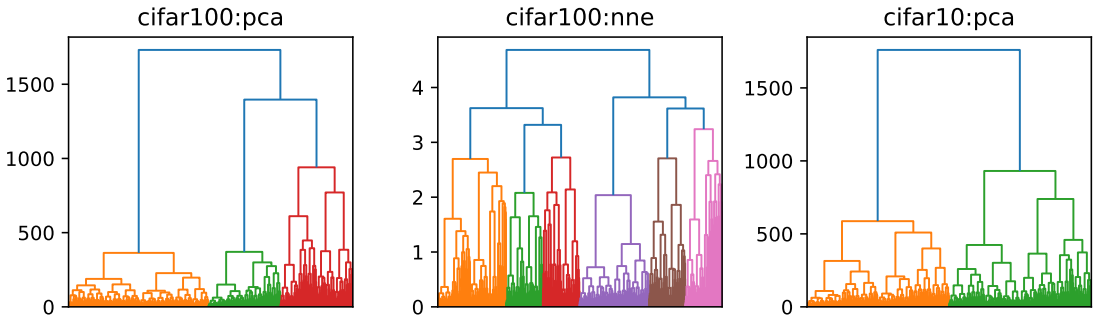
In this section, we describe the techniques we study to reduce the dimensionality of the datatsets before clustering.

Principal Component Analysis (PCA): We use incremental PCA from the SciPy library. As explained in Chapter 5.1.3.2, we use the Explained Variance Ratio (EVR) to identify the number of components we should use. Figure 5.7 shows the cumulative sum of the EVR as we add more components, for the CIFAR-10 and CIFAR-100 datasets. We aim for a cumulative EVR value of 0.95 (or 95%) to ensure that our subsequent clustering captures as much of the information in our datasets as possible. We denote this level with the dotted line in Figure 5.7. For both datasets, we see that we reach an EVR of ~ 0.80 with just 25 components. The increase in EVR then quickly plateaus and we need 200 components to reach our target EVR of 0.95. We therefore set $N = 200$ for PCA for the rest of our analysis.

Neural Network Embedding (NNE): For generating embeddings using Neural Networks, we use the technique proposed by Znalezniak et al. [270]. First, they train a ResNet-50 model to achieve high classification accuracy on the target dataset. They then remove the final fully-connected layer of the network. For example, for CIFAR-10, this is the final layer that has 10 outputs, one for each output class. The trimmed ResNet-50 network has an output size of 512, which they then cluster using Hierarchical Agglomerative Clustering.



(a) Comparison of distance methods for NNE on CIFAR-10.



(b) Ward clustering with PCA and NNE for CIFAR-100 and CIFAR-10.

Figure 5.8: Dendograms of PCA and NNE data for CIFAR-10 and CIFAR-100.

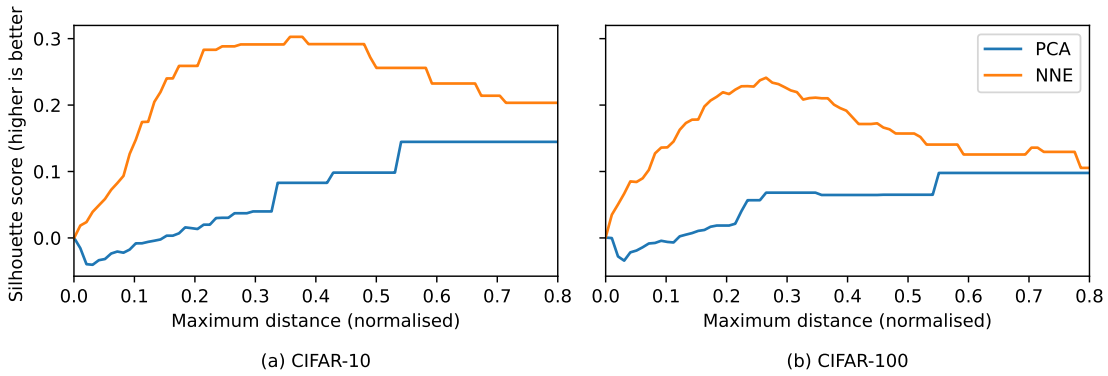
5.3.2 Data clustering

As we use Hierarchical Agglomerative Clustering (HAC), we must specify a method of determining the distance between clusters. We explore three methods: centroid, median and Ward. For all methods, we use the Euclidean or L_2 distance as our metric. To identify the best method, we plot the dendograms after clustering with each.

Figure 5.8a shows the dendograms for NNE on CIFAR-10 for each method. We color the clusters based on their relative distances; clusters which are apart by at least half the maximum distance are colored differently. This provides an easy way to visually see the separation between clusters, as dendograms with more colors at the bottom are better separated. We see that both centroid and median perform very poorly as there are no clearly separable clusters. Thus, both these figures only have a single color. In contrast, Ward clustering performs much better and we see the gradual separation into more clusters as we move down the figure as expected. We see similar results for the CIFAR-100 dataset as well. Based on this, we continue with the Ward method for our work.

We now compare PCA vs. NNE data for clustering quality. Figure 5.8b shows the dendograms using Ward clustering for NNE and PCA on both datasets. Comparing PCA and NNE for CIFAR-10 (the right dendograms in Figures 5.8a and 5.8b), we see that NNE better separates the clusters. We can also see this from the coloring, as the NNE graph has many more colors compared to the PCA graph. We see a similar result for CIFAR-100 (left and middle in Figure 5.8b), which tells us that NNE using Ward clustering performs the best.

Number of clusters: The next step is to identify the number of clusters we should use. This requires us to find the best point to 'cut' the dendogram to identify clusters. We do this using



(a) CIFAR-10 (b) CIFAR-100

Figure 5.9: Silhouette scores when clustering PCA and NNE data using ward HAC.

the *fcluster* function in SciPy, which performs HAC. We start by calculating the minimum (d_{min}) and maximum distance (d_{max}) within the entire dendogram. We then calculate 100 steps between d_{min} and d_{max} and compute the number of resulting clusters at each step. This allows us to check the entire range of distances to identify the optimum value.

To determine the quality of the clustering at each distance, we use the Silhouette score, as described in Chapter 5.1.3.3. To validate our observation that NNE outperforms PCA, we compute the Silhouette score for data from both these dimensionality reduction techniques. Figure 5.9 shows the plot of Silhouette scores for CIFAR-10 and CIFAR-100. We show the scores when performing ward clustering on both the PCA and NNE data. We only show values up to 0.8 on the x-axes, as the values are the same in the range $[0.8, 1.0)$. Distances going along the x-axis in Figure 5.9 correspond to going from the bottom to the top in Figures 5.8a and 5.8b. For PCA, we see that the silhouette scores initially become negative, which indicates that inputs are in the wrong cluster. Also, as we increase distance, the size of the clusters also increases. Thus, the scores for PCA only increase when we reach large clusters as the data is now more likely to be in the correct cluster.

In contrast, NNE achieves higher scores for both datasets. This shows us that the neural embedding is better able to capture the similarity between inputs in the clusters. We see that the scores start off low at d_{min}, as we cannot cluster any points yet. Then, as we increase the distance, we start clustering more points together, which are close to each other, resulting in a higher score. However, after a certain point, the score begins to drop again, as we are now clustering inputs which should belong to different clusters. Therefore, we cluster the datasets at the distances with the highest Silhouette scores (i.e., $0.36d_{max}$ for CIFAR-10 and $0.28d_{max}$ for CIFAR-100). These correspond to Euclidean distances of 1.23 and 1.48 for CIFAR-10 and CIFAR-100, respectively.

5.3.3 Effect on random sampling

In order for EMPATIC to be effective, we must form batches with inputs which are close together. To see how our technique achieves this, we now show the impact of loading batches of clustered vs. unmodified data. As described in Chapter 5.2.2, we implement a custom dataloader to support loading clustered inputs. We compare the distance between every pair of inputs in each batch, using our custom vs. the regular dataloader. For this experiment, we set the batch size to 4, which is the default size of clusters we use in our evaluation. Thus, we measure the distances within the

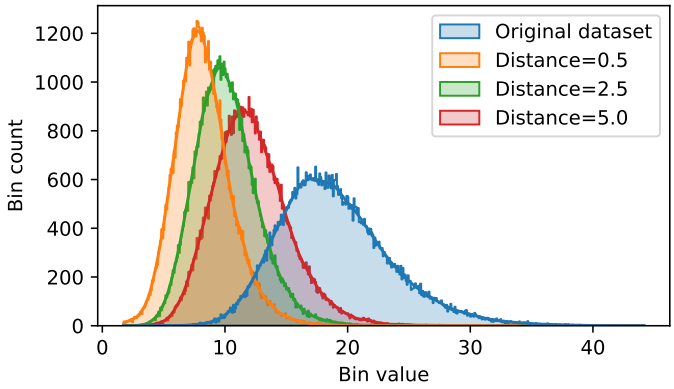


Figure 5.10: Distribution of pairwise distances in each batch, with and without using clustering.

group of inputs which our technique will compute deltas for.

Figure 5.10 shows the distribution of pairwise distances when using the regular vs. custom dataloader. We also sweep the distance used for clustering, from 0.5 to 2.5 to 5.0. For the regular dataloader, inputs are sampled from the entire training set for each batch. Thus, we see a wide distribution of distances (mean=18.96, max=44.14). We see that a distance of 0.5 yields a much narrower distribution (mean=8.23, max=21.98). Then, as we increase the distance to 2.5 and 5.0, the distribution gradually widens and shifts further to the right. This shows that EMPATIC is effective at reducing the distance between inputs, compared to using the regular dataloader.

5.4 Evaluation

We now evaluate he effectiveness of EMPATIC, for accelerating adversarial training. We begin by detailing our methodology (Chapter 5.4.1). We then analyze the impact of EMPATIC on network accuracies (Chapter 5.4.2). Finally, we study the performance overhead of EMPATIC, compared to training with $FP16$ (Chapter 5.4.3).

5.4.1 Methodology

We use two systems in our evaluation: 1) a desktop system for short experiments and 2) a server system with multiple GPUs for longer runs. Table 5.2 provides the specifications for both systems. For both GPUs, we expect a theoretical speed-up of 8× for $FP16$ vs. $FP32$ (as we showed in Figure 5.3). We use PyTorch 1.12 for all our experiments [212]. We evaluate our approach using three networks commonly used in prior work: ResNet-18, ResNet-50 and WideResnet-28_10 [209]. We report robustness accuracy for AutoAttack [46].

For training hyperparameters, we use the recommendations made by Pang et al. [209]. They perform a comprehensive evaluation of the impact of several hyperparameters on the effectiveness of adversarial training. We tabulate these hyperparamaters in Table 5.3, along with the attack settings we use during adversarial training. All the models are trained for 150 epochs with the learning rate decaying by a factor of 0.1 at 100 and 125 epochs, respectively.

For networks trained with $FP32$ we perform training with no modifications to the baseline PyTorch implementation. For $FP16$, we use the *mixed-precision training* functionality of PyTorch. For $FP8$ training, as we do not have hardware with $FP8$ support, we emulate $FP8$

Table 5.2: Configuration of systems used in our evaluation.

System	Desktop	Scinet
CPU		
Model	Intel i7-6700	IBM Power 9
Cores	8	32
Frequency	3.40 GHz	2.80 GHz
GPU		
Model	Nvidia GeForce 2080 Ti	4 x Nvidia V100-SXM2
Generation	Turing	Volta
CUDA cores	4,352	5,120
GPU memory	11 GB GDDR6	32 GB HBM2
CUDA version	11.2	11.3
System		
System Memory	16 GB	256GB
OS	Ubuntu 20.04.6 LTS	Red Hat Enterprise Linux 8.2

Table 5.3: Hyperparameters for model training and attack parameters for adversarial training.

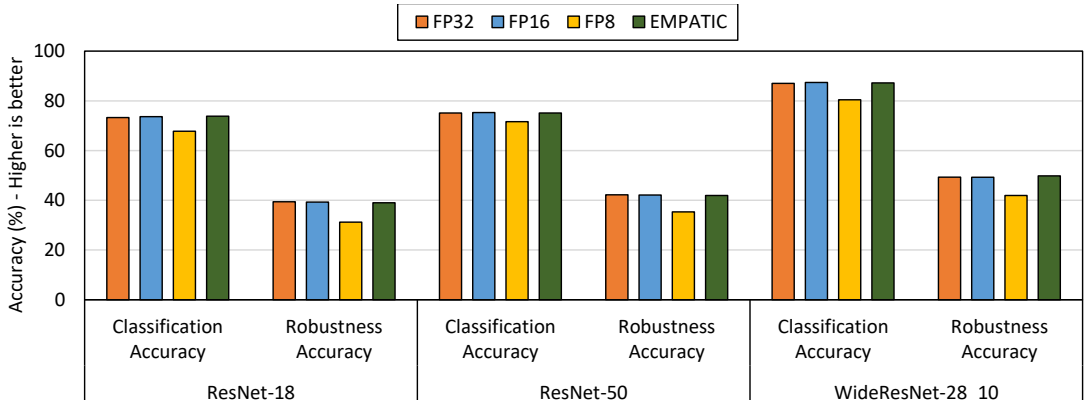
Parameter	Setting	Parameter	Setting
Batch size	128	Attack	PGD
Optimizer	SGD	Norm	L_{inf}
Weight decay	5×10^{-4}	Steps	10
Momentum	0.9	Alpha	$2/255$
Initial learning rate	0.1	Epsilon	$8/255$

similar to prior work [175, 182]. For $FP8$ training and EMPATIC, we use $E4M3$ during the forward pass and $E5M2$ for the backward pass, also matching prior work [199]. We employ adaptive loss scaling for $FP16$, $FP8$ and EMPATIC. For EMPATIC, we vary the cluster size between 4, 8 and 12 to understand the impact of this parameter on network performance.

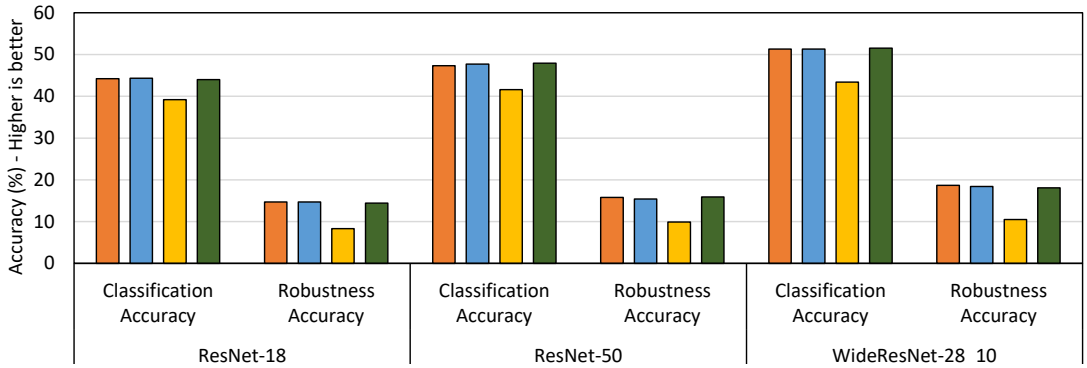
5.4.2 Network accuracy

We begin by presenting the impact of EMPATIC on the classification and robustness accuracies. Figure 5.11 shows our results when training using the $FP16$ and $FP8$ formats as well as using EMPATIC. For all the networks we study, we see that training using $FP16$ obtains $\pm 1\%$ of the classification and robustness accuracies of $FP32$. However, even with scaling, $FP8$ trained networks are unable to achieve the same accuracies as $FP16$. In contrast, we see that EMPATIC is able to achieve comparable accuracies to $FP16$. Thus, EMPATIC is able to effectively use the reduced range of $FP8$ during adversarial training. We do not show the impact of changing the cluster size as it does not affect accuracy.

Figure 5.12 shows the percentage of gradient values which underflow $FP8$. We collect statistics for every Conv2d layer per network during each training epoch. We only measure the gradients of inputs computed during the attacks during adversarial training, as this is the step which we



(a) CIFAR-10.



(b) CIFAR-100

Figure 5.11: Classification and robustness accuracies for the networks we study.

aim to speed-up using EMPATIC. We calculate the number of gradients which are too small to be represented using the $FP8$ format as a percentage of the total number of gradients for each layer. We then plot the statistics for every Conv2d layer for each epoch. Each dark line represents the average while the light-colored bands around each line show values with a 95% confidence interval. For EMPATIC alone, we show statistics for the deltas computed rather than all the gradient values. We do this because we only convert the deltas to $FP8$ and not the entire gradient, as is done for the other cases.

We first analyze the behaviours which are common for all the cases we study. First, we see a noticable 'jump' for all the lines at 100 epochs. This is because our learning rate is reduced to 0.01 from the initial value of 0.1 at this point. The learning rate is once again adjusted to 0.001 at 125 epochs, but this is only noticeable for ResNet-50 running CIFAR-10. We believe this jump is because of the multiplication of the gradients with the learning rate during training. Thus, a smaller learning rate leads to smaller values being produced through the network during the backward pass. These smaller values are then more likely to underflow the narrow range of $FP8$ and we see an increase in the underflow rate when this happens.

Looking at each datatype in turn, we see that training with $FP16$ has the most values which underflow the $FP8$ format. This is expected as values which underflow $FP8$ can still be represented in $FP16$ and the networks can successfully converge. However, when we train using $FP8$ we see significantly fewer values with underflow this format. This tells us that the model is adjusting to the values which are being lost due to underflow. Despite this, the percentage of

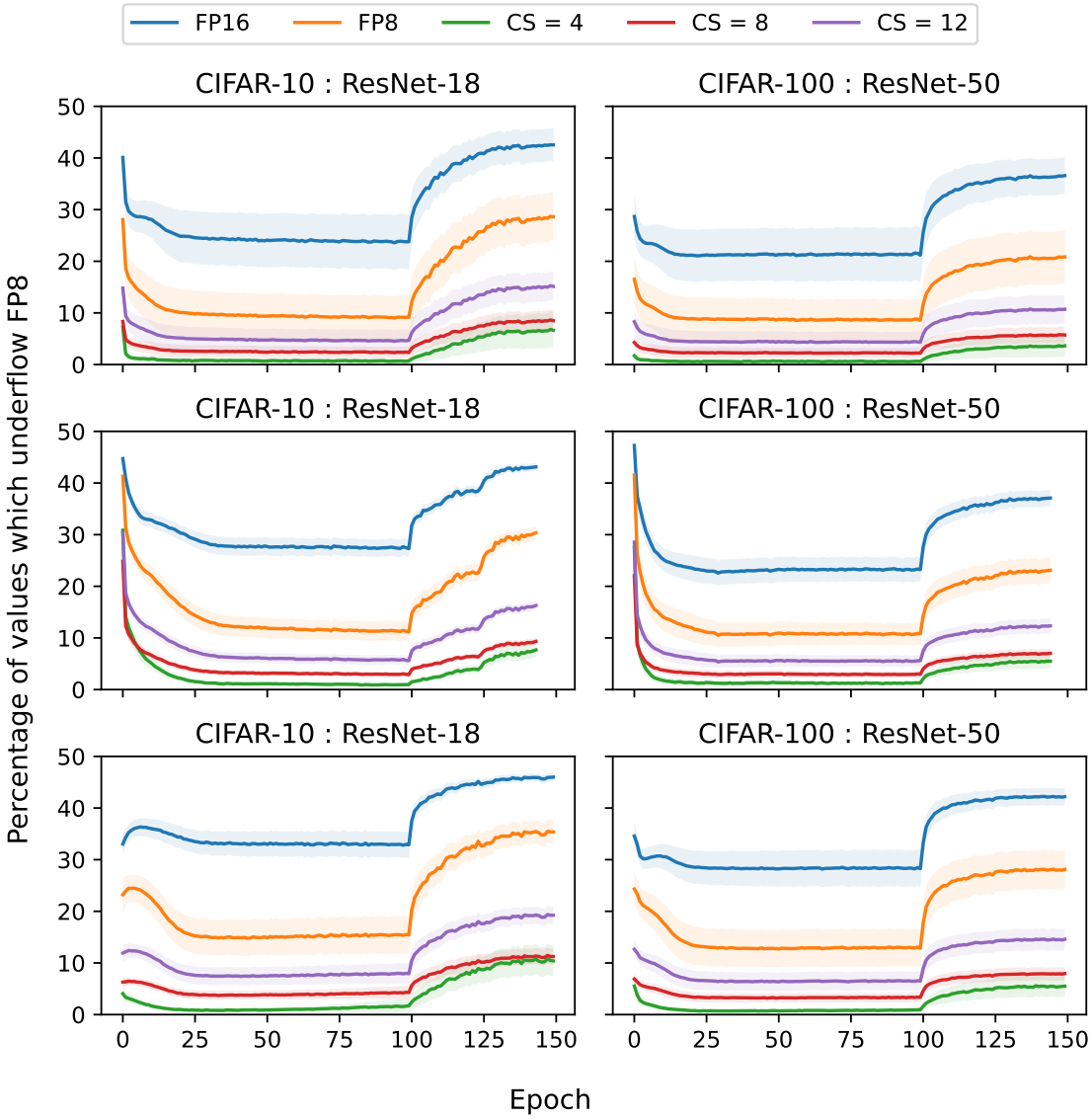


Figure 5.12: Percentage of gradients per epoch that underflow FP8.

values which underflow during $FP8$ training is still high. This may explain why training with $FP8$ fails to match the accuracy of models trained using $FP16$.

Finally, for EMPATIC, we see the lowest percentage of underflow values. Note that for EMPATIC, we show the percentage of deltas (and not gradients) which underflow. Unlike the network accuracies, we see than changing the cluster size does affect underflow. For a cluster size of 4 (CS = 4 in Figure 5.12), we see the lowest values of underflow. Then, as we increase cluster size to 8 and 12, we see that our underflow rate increases somewhat. This is expected as we are computing the difference across more inputs when we increase the cluster size. Thus, inputs are more likely to be farther apart with a larger cluster size. The deltas computed in that case are more likely to be larger. Since we perform adaptive loss scaling, this means that our scale factor is smaller to avoid any values overflowing $FP8$. As a consequence, we are more likely to lose information in the lower bits when values are stored in $FP8$ format. Despite this, we see that even a cluster size of 12 has far less underflow than $FP16$.

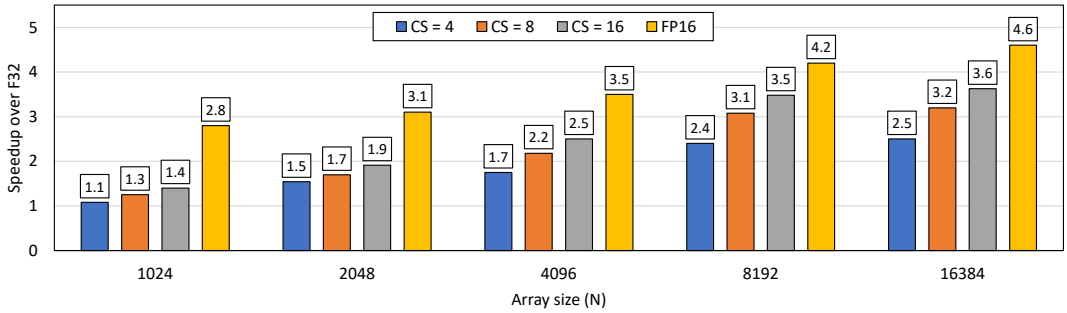


Figure 5.13: Performance impact of EMPATIC for matrix-multiplication.

5.4.3 EMPATIC overhead

We now analyze the overhead of EMPATIC. As mentioned earlier, we do not have access to the latest GPUs which support $FP8$ computation. Both of the GPUs we use support computations using $FP32$ and $FP16$. As we described in Chapter 5.2.1.1, we keep the weights and activations in $FP16$ and only 'convert' them to $FP8$ before we use them. In this section, we instead keep the values in $FP32$ and perform clustering using $FP16$. We then compare the time to train the networks using the $FP32$ and $FP16$ formats as well as our modified method. This allows us to compare the runtime of EMPATIC vs. training entirely using $FP32$ or $FP16$.

Matrix multiplication: We begin with the simple experiment of multiplying two randomly generated $N \times N$ matrices. We generate these two matrices using the CPU and copy them to GPU memory each time. Thus, each experiment comprises the time to: 1) read the matrices from GPU memory, 2) perform the matrix multiplication and 3) write the results back to GPU memory.

Figure 5.13 shows the results of our experiment. We vary N from 1024 to $16,384$, to understand how our results vary with larger matrices. Once again, we study three cluster sizes: 4, 8 and 12. We plot the speedup for each case compared to the baseline $FP32$ case.

As we increase N, we see that $FP16$ sees a significant improvement in throughput. For large matrix sizes (such as $N = 16,384$), this can be as high as $4.6\times$. However, this is still far from the peak theoretical speedup for our GPUs of $8\times$. This gap is due to the overhead of reading matrices from memory and the time taken to invoke GPU kernels.

For EMPATIC, for small matrices, we see that the overhead of our technique dominates and we only see a modest speedup. However, for larger matrix sizes, EMPATIC begins to approach the speedup of $FP16$. For large matrices (e.g., $N = 16,384$), we achieve $\sim 60-80\%$ of the performance of $FP16$, depending on the cluster size we use.

Network latency: We now detail the performance impact of EMPATIC on the networks we study. We present the end-to-end time to train each network for 150 epochs. Once again, for EMPATIC, we show results when the values are stored using $FP32$ and we perform clustering using $FP16$. Figure 5.14 shows the speedup of $FP16$ and EMPATIC (using different cluster sizes), normalized to the latency of $FP32$. Compared to Figure 5.13, we see that $FP16$ achieves more modest speed-ups from $1.7\times$ to $2.7\times$. We believe this is because of operations which do not benefit from the smaller datatype, such as the overhead of invoking CUDA kernels, disk

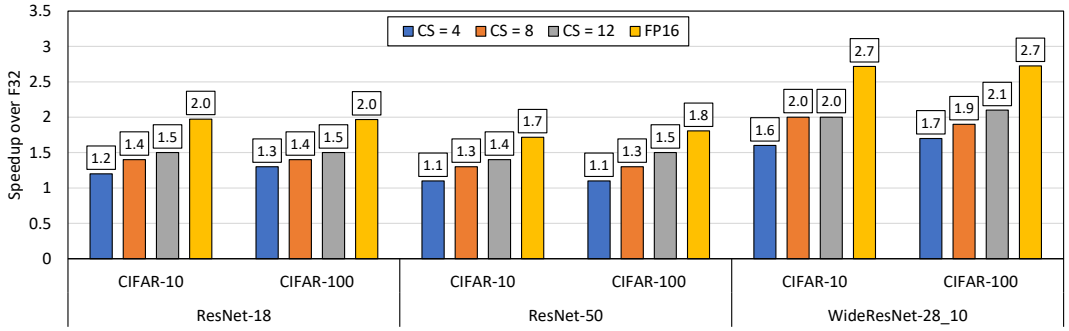


Figure 5.14: Performance impact of EMPATIC for networks.

I/O etc. PyTorch also relies on extensive synchronization between the CPU and GPU to ensure correctness. This adds further overhead to the native PyTorch code we use to perform $FP16$ training, as there is a synchronization delay after most kernel calls.

For EMPATIC, we see consistent speed-ups over $FP32$, ranging from $1.1\times$ for $CS = 4$ for ResNet-50 to $2.1\times$ for $CS = 12$ for WideResNet-28_10. However, we see that EMPATIC does not lag behind $FP16$ as much as in our previous matrix-multiplication experiment. We are able to 'close the gap' with the $FP16$ training time with our custom CUDA implementation which avoids much of the synchronization overhead introduced by using the native PyTorch functions. For matrix-multiplication, we only call a single PyTorch function (i.e., $matmul()$), which does not introduce the overheads we see when running the networks. These experiments show that running EMPATIC on hardware which supports $FP8$ will provide a further speed-up over $FP32$, while still achieving high classification and robustness accuracy.

5.5 Related work

In this section, we present related work to our technique. We begin with prior works which focus on accelerating adversarial training, through algorithmic modifications. We then describe prior works which use 8-bit floating-point for ML training. Finally, we list other approaches for data clustering, in addition to the hierarchical agglomerative clustering technique we use in EMPATIC.

5.5.1 Accelerating adversarial training

Adversarial training imposes a significant slowdown during training, as the attack are run in steps and each step performs a full forward and backward pass through the network. Thus, a 10-step PGD attack imposes a roughly $10\times$ increase to the model training time. Therefore, several techniques have been proposed to reduced the latency overhead of adversarial training. Early work modified training to run the same minibatch m times in a row [228]. This allows for multiple updates to the image during each epoch. Wong et al. proposed using the single-iteration FGSM attack [253]. However, follow on work showed that both these techniques do not significantly improve robustness [7]. Gupta et al. propose delayed adversarial training where they train the model for several epochs with natural inputs before switching to adversarial inputs [100].

They show that switching as late as 80 epochs allows the model to still achieve comparable classification and robustness accuracy as training with AT from the very start.

Another approach is to identify a reduced set of the training inputs and only perform training on this set. Bullet Train shows that adversarial training only benefits inputs which are close to the decision boundary and is redundant on inputs which are far from this boundary [118]. Thus, they prioritize adversarial training on this 'near-boundary' subset to speed up training by $2\times$ without any change in classification or robustness accuracy. Dolatbadi et al. identify a 'coreset' of the training inputs per epoch and only perform adversarial training on these inputs [54]. They see a $2-3\times$ speedup with only a minor loss of accuracy. Li et al. propose GRAD-MATCH, which finds the data subset whose gradients most closely match those of the full training data [157] which yields a $2-3\times$ speedup with only a minor loss of accuracy. However, these prior works do not investigate clustering or specifically targets $FP8$ hardware. Thus, EMPATIC can be used in conjunction with these existing approaches to further accelerate adversarial training.

5.5.2 8-bit floating point training

There has been some prior work on using modified $FP8$ formats for training. Cambier et al. propose a new $FP8$ format which stores multiple values together [28]. This new format uses two extra factors for 'scaling' and 'shifting' values to the optimal range of $FP8$ [28]. However, this new format requires custom hardware which is not discussed in this work. Wortsman et al. explore the use of 8-bit integer to speed up training of large language models [254].

Another avenue for enabling $FP8$ training is to use a different exponent bias for each layer. As the exponent bias controls the range of values that a given format can represent, this allows for tuning the range to manage underflow. This is known as 'hybrid floating point' (HFP). Noune et al. perform an analysis of $FP8$ training and observe that the choice of exponent bias significantly affects accuracy [199]. They show that they can match the accuracy of $FP16$ by selecting the appropriate exponent bias for $FP8$. However, as we cannot change the bias on GPUs, we are not able to do such tuning. They also only test using ResNet-32 on CIFAR-100 so it is difficult to know if their observations extend to other networks and datasets. There are also hardware implementations which use the HFP-8 format [**agrawal20219**, 242]. However, these approaches require a different format of 8-bit float from the ones supported by modern GPUs. Thus, unlike EMPATIC, these techniques cannot be used to accelerate robust training on GPUs.

5.6 Conclusion

In this chapter, we aim to speed-up adversarial training on GPUs. Due to the iterative nature of adversarial attacks, they impose a significant performance overhead during training. We propose EMPATIC to accelerate adversarial training high-throughput $FP8$ computations on GPUs. To overcome the limitations of prior work when using $FP8$, we first cluster the training data to identify inputs which are close to each other. We then create mini-batches where nearby inputs are run contiguously in groups. At training time, we split these groups into a base and compute deltas between this base and the other inputs in the group. This yields deltas which have a smaller range than the original inputs, owing to our clustering. We then perform computations using

these deltas using $FP8$ and see significantly less overflow than prior approaches. EMPATIC can therefore utilize GPU support for high-throughput $FP8$ computations to train robust ML models.

6

CONCLUSION

The importance of machine learning and its use in our daily lives is rapidly increasing. Ensuring the security and privacy of ML systems is of paramount importance. At the same time, ML is being deployed on a range of computing platforms, from low-power IoT devices to high-performance GPUs. We have also seen an explosion in the number of accelerators designed specifically to accelerate ML workloads. The focus of this thesis has been to improve the security and privacy of machine learning (ML) applications. To that end, each of the contributions in this thesis targets a different hardware platform which runs ML. We now summarize these three contributions and provide suggestions for future avenues of research.

6.1 FARO

ML models are increasingly used in low-power IoT devices. As models deployed on such devices are susceptible to side-channel attacks, we require a way to secure these models. These attacks require operations to be performed in the same order each time, to allow for collected power traces to be averaged. Prior works propose shuffling the order of operations to prevent these attacks. However, we show that performing shuffling in software leaks side-channel information which can be used to undermine the security benefits of shuffling. We also show that software shuffling can significantly slowdown the model's inference time.

To securely and efficiently perform shuffling, we present FARO. FARO comprises hardware – added as a functional unit within the CPU – to shuffle arbitrary numbers of items (N). We are the first to support shuffling arbitrary values of N in hardware, as prior works only work when N is a power of two. FARO secures the weights of ML models running on IoT devices while adding just 0.56% latency, 2.46% area and 3.28% power overheads.

6.1.1 Future directions

We hope that FARO will motivate researchers to explore other areas where shuffling can be beneficial, such as the applications we mentioned in Chapter 3.6.1. With FARO, we focused on securing existing ML models. However, we believe that developing the model with shuffling in mind, can be an interesting research direction. One approach could be to explore models with fewer layers but more channels per layer. As the security benefit of FAROimproves with size, shuffling layers with more channels can further improve security.

Another approach would be to explore automatic model architecture generation. For example, 'Neural Architecture Search' (NAS) has been proposed to develop models which are more robust against adversarial attacks [246]. We propose extending this to generate models which can further benefit from shuffling. Models generated with NAS have significantly more 'branching'; there are multiple sequences of layers which must be run in parallel. Since these parallel group of layers can be run in any order, we can shuffle them to make side-channel attacks more difficult. This gives us another dimension to shuffle using FARO, to further improve side-channel security. Thus, NAS could be used to explore creating models that are robust against both adversarial and side channel attacks.

6.2 AESIR

The popularity of ML has resulted in many accelerators specifically designed for ML models. However, existing designs cannot run security-critical ML algorithms. Two important algorithms which current ML accelerators cannot support are: 1) Differentially private training and 2) Adversarially robust inference. Both algorithms require random values – sampled from specific distributions – to guarantee security and user privacy. However, existing designs for generating such 'noise' directly in hardware impose significant overheads. Furthermore, current approaches also leak side-channel information that can undermine their security [131, 186].

To enable ML accelerators to support these crucial algorithms, we propose AESIR. AESIR pre-computes the required noise ahead of time and stores these values in plentiful off-chip DRAM, along with the model weights. Then, when we require noise, we can *randomly sample* values from this stored list. This allows us to support adding noise from different distributions, while also avoiding the side-channel information leakage that plagues on-chip noise generation. AESIR enables noise addition while incurring 23× lower area and 40× lower energy compared to producing noise directly on chip.

6.2.1 Future directions

AESIR is a technique to enable noise addition from arbitrary distributions in hardware. While AESIR adds $< 5\%$ DRAM memory overhead, future work could explore ways to compress the noise values so even more points could be stored for greater security.

We also limit our study to adding Gaussian or Laplace noise, as done in prior work. This is because these distributions are well studied and there are established ways to produce noise from these distributions in both software and hardware. However, AESIR is flexible and adding noise from any distribution would incur the same overhead. Thus, another avenue for future research could be to explore other – possibly more complex – distributions to understand if they would offer greater security or other benefits.

6.3 EMPATIC

Finally, our last contribution aims to speed up adversarial training on modern GPUs. Adversarial training makes models more robust by training the model on inputs which have already been

attacked. Thus, we perform the attacks during training so that the model learns to correctly classify attacked inputs. However, adversarial training incurs a significant performance overhead, due to the iterative nature of attacks. Each attack iteration requires a full pass – forward and backward – through the network.

We describe EMPATIC, a technique to speed up adversarial training on modern GPUs which support high-throughput 8-bit floating-point computations. Many prior works have leveraged GPU support for narrow-width floating point formats (such as 16-bit floating point) to accelerate training. However, these approaches struggle to extend to the extremely narrow range of 8-bit formats. To overcome this, we first perform clustering on the training set to create 'groups' of inputs which are near each other. Within each group, we maintain one input as a 'base' input and compute the difference for the other inputs to the base input. By doing so, we obtain deltas which (due to our clustering) have a smaller range than the original training inputs. We then perform computations using these deltas with an 8-bit datatype. We show that EMPATIC effectively leverages GPU support for high-throughput 8-bit operations, without suffering the drawbacks of prior approaches.

6.3.1 Future directions

With EMPATIC, our goal was to make better use of GPU support for $FP8$. Another major avenue for GPU hardware is to support sparse computations. Modern GPUs provide increased throughput if 50%+ of values are 0 [5, 203]. Future work could investigate using this capability to further speed up adversarial training. Specifically, due to our clustering, we obtain deltas between similar inputs which may be more likely to have many 0s. If this is the case, we can 'sparsify' (i.e., convert the values to a sparse data representation) the deltas and obtain a speed-up when running them on a GPU.

Appendices

A

FOLDING BATCHNORM DURING INFERENCE

In this appendix, we describe how the computation of a Batch Normalization (BatchNorm) layer can be combined with the preceding convolutional layer during inference.

Operation of BatchNorm: The first step of BatchNorm is computing the mean and variance of the activations in each batch. For inputs x_i, we compute the mean (μ) and variance (σ^2) for each batch as:

$$\mu_b = \frac{1}{B}\sum_{i=1}^{B} x_i \tag{A.1}$$

$$\sigma_b^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_b)^2 \tag{A.2}$$

We can then compute the normalized values $\hat{x}_i$ as:

$$\hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \varepsilon}} \tag{A.3}$$

In Equation A.3, we add a small positive value ε for numerical stability, when σ_b^2 is small. All the $\hat{x}_i$ values now have a mean of 0 and a variance of 1. However, forcing inputs across all batches to have the same mean and variance can adversely affect performance. Thus, we add two parameters (γ and β) per mini-batch, which are learned during training. This allows the network to adjust the relative normalization across mini-batches and maintain high accuracy. The final outputs of the BatchNorm layer are computed as:

$$y_i = \gamma\hat{x}_i + \beta \tag{A.4}$$

Folding BatchNorm BatchNorm plays a key part during network training. However, during inference, we know all the parameters of the BatchNorm layer ahead of time. Thus, applying BatchNorm during inference merely becomes applying a linear transform to the data. We can therefore 'fold' the BatchNorm during inference with the preceding convolutional layer. We start by replacing $\hat{x}_i$ in Equation A.4 to get:

$$y_i = \frac{\gamma(x_i - \mu_b)}{\sqrt{\sigma_b^2 + \varepsilon}} + \beta \tag{A.5}$$

Since x_i is the output of the preceeding convolutional layer, we can rewrite this as:

$$x_i = W^p \cdot x_i^p + b^p \tag{A.6}$$

where W^p and b^p are the weights and biases of the previous layer.

Replacing Equation A.6 in Equation A.5, we get:

$$y_i = \frac{\gamma\left[(W^p \cdot x_i^p + b^p) - \mu_b)\right]}{\sqrt{\sigma_b^2 + \varepsilon}} + \beta \tag{A.7}$$

We can now rewrite Equation A.7 as:

$$y_i = W' \cdot x_i^p + b' \tag{A.8}$$

where, W' and b' are:

$$W' = W^p\left(\frac{\gamma}{\sqrt{\sigma_b^2 + \varepsilon}}\right) \tag{A.9}$$

$$b' = \frac{\gamma}{\sqrt{\sigma_b^2 + \varepsilon}}(b^p - \mu) + \beta \tag{A.10}$$

As all the parameters in Equations A.9 and A.10 are known during inference, we can compute W' and b'. We can then use these directly in the preceding convolutional layer to obtain the benefit of normalization, with no additional overhead.

B

FARO : LAYER ANNOTATIONS

In this Chapter, we provide code for additional layers which implements FARO. Algorithm 8 shows the modified code for 2D convolutional layers, while Algorithm 9 shows code for a 2D max pooling layer.

Code Snippet 8: Convolutional layer with FAROfunctions added.

```
1  load_bank(BANK0,M)
2  load_bank(BANK1,N)
3  load_bank(BANK2,R)
4  load_bank(BANK3,C)
5  for oc = 0; oc < M; outch++ do
6      r_oc = get_next_iteration(BANK0)
7      for ic = 0; ic < N; inch++ do
8          r_ic = get_next_iteration(BANK1)
9          for r = 0; r < R; row++ do
10             r_r = get_next_iteration(BANK2)
11             for c = 0; c < C; col++ do
12                 r_c = get_next_iteration(BANK3)
13                 for i = 0; i < K; i++ do
14                     for j = 0; j < K; j++ do
15                         sum[r_oc][r_r][r_c] += input[r_ic][S×r_r+i][S×r_c+j] ×
                             weight[r_oc][r_ic][i][j]
16                     end
17                 end
18                 sum[r_oc][r_r][r_c] += bias[r_oc]
19                 output[r_oc][r_r][r_c] = actFunc(sum[r_oc][r_r][r_c])
20             end
21         end
22     end
23 end
```

Code Snippet 9: 2×2 max pooling layer with FAROfunctions added.

```
1  load_bank(BANK0,N)
2  load_bank(BANK1,R)
3  load_bank(BANK2,C)
4  for ch = 0; ch < N; i++ do
5      r_ch = get_next_iteration(BANK0)
6      for r = 0; i < R; r++ do
7          r_r = get_next_iteration(BANK1)
8          for c = 0; j < C; c++ do
9              r_c = get_next_iteration(BANK2)
10             output[r_ch][r_r][r_c] = max(input[r_ch][r_r][r_c], input[r_ch][r_r+1][r_c],
                 input[r_ch][r_r][r_c+1], input[r_ch][r_r+1][r_c+1])
11         end
12     end
13 end
```

C

EMPATIC : CLASS LABELS WITH CLUSTERING

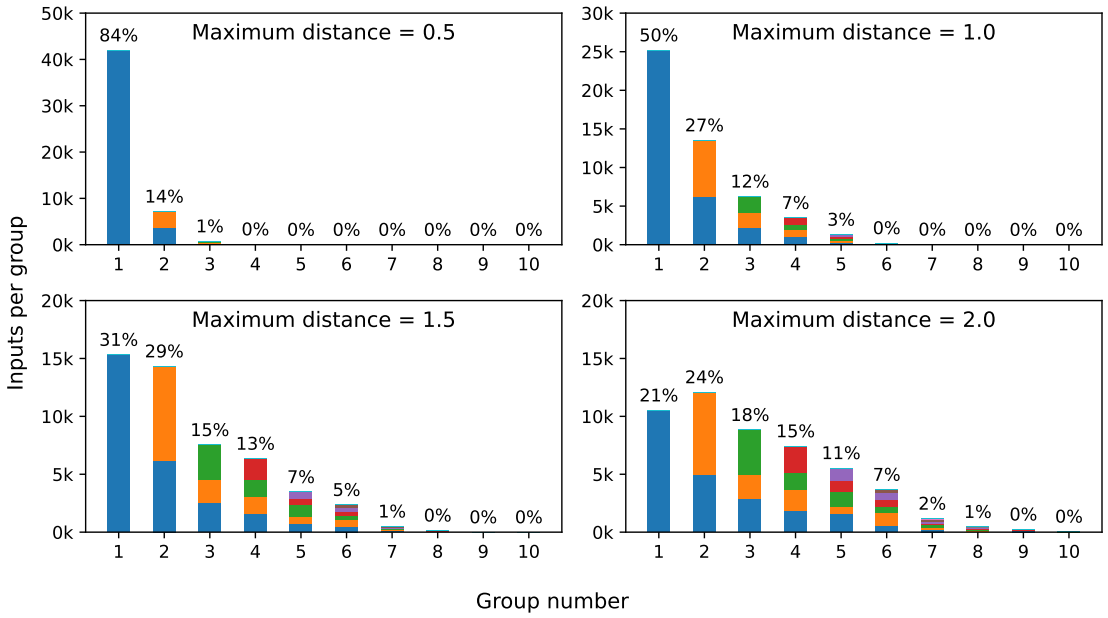


Figure C.1: Distribution of class labels by varying clustering distance for CIFAR-10.

We now analyze how the clustering we perform in Chapter 5 correlates with the ground truth class labels. We do so with the question *do clusters comprise inputs from the same or different classes?* Once again, we consider neural embedding (NNE) data, clustered using hierarchical agglomerative clustering with the ward method. Figure C.1 shows how varying the clustering distance (from 0.5 to 2.0) affects the mix of class labels in each cluster for the CIFAR-10 dataset. We vary the distances around our optimum distance of 1.23 for the CIFAR-10 dataset. We only analyze the CIFAR-10 dataset, which has 10 classes. As CIFAR-100 100 classes it is difficult to represent visually. For each distance we evaluate, we compute the number of classes in each cluster. We then split these into 'groups' where group n has n labels in each cluster. To further visually represent this information, every color in a bar represents a different class. Thus, group 1 only has a single color, group 2 has two colors and so on.

For a distance of 0.5, 84% of the clusters have inputs from a single class. This is expected, as our clusters only encompass a small area, which is more likely to only have inputs from a

single class. As we increase the distance, we see clusters have more classes. At a largest distance we study (i.e., 2.0), we see that only 21% of clusters have a single class. We see that even with clustering, all but the smallest regions have inputs belonging to different classes.

BIBLIOGRAPHY

[1] J. M. Abowd, "The US census bureau adopts differential privacy," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[2] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[3] M. Ahmad, P. M. Khan, and M. Z. Ansari, "A simple and efficient key-dependent s-box design using fisher-yates shuffle technique," in *Proceedings of the 2nd International Conference on Recent Trends in Computer Networks and Distributed Systems Security (SNDS)*, 2014.

[4] M. Aitsam, "Differential privacy made easy," in *Proceedings of the International Conference on Emerging Trends in Electrical, Control, and Telecommunication Engineering (ETECTE)*, 2022.

[5] *AMD Instinct MI300X Platform*, 2023. [Online]. Available: https://www.amd.com/en/products/accelerators/instinct/mi300/platform.html.

[6] M. Andriushchenko, F. Croce, N. Flammarion, and M. Hein, "Square attack: A query-efficient black-box adversarial attack via random search," in *Proceedings of the European Conference on Computer Vision*, 2020.

[7] M. Andriushchenko and N. Flammarion, "Understanding and improving fast adversarial training," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[8] F. Antognazza, A. Barenghi, and G. Pelosi, "Metis: An integrated morphing engine cpu to protect against side channel attacks," *IEEE Access*, vol. 9, 2021.

[9] Apple, *Learning with privacy at scale*, https://docs-assets.developer.apple.com/ml-research/papers/learning-with-privacy-at-scale.pdf, accessed on 2022-06-08, 2017.

[10] A. Araujo, L. Meunier, R. Pinot, and B. Negrevergne, "Robust neural networks using randomized adversarial training," *arXiv preprint arXiv:1903.10219*, 2019. [Online]. Available: https://arxiv.org/abs/1903.10219.

[11] *Arm architecture reference manual*, ARM Limited, 2019. [Online]. Available: https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf.

[12] V. Arribas, S. Nikova, and V. Rijmen, "Vermi: Verification tool for masked implementations," in *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018.

[13] *Artificial intelligence and machine learning projects are obstructed by data issues*, Accessed on Dec. 7, 2023. [Online]. Available: https://cdn2.hubspot.net/hubfs/3971219/Survey%20Assets%201905/Dimensional%20Research%20Machine%20Learning%20PPT%20Report%20FINAL.pdf.

[14] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow, J. H. Ahn, and N. S. Kim, "Near-dram acceleration with single-isa heterogeneous processing in standard memory modules," *IEEE Micro*, vol. 36, no. 1, 2016.

[15] M. Bafna, J. Murtagh, and N. Vyas, "Thwarting adversarial examples: An L_0-robust sparse fourier transform," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[16] M. Bakiri, C. Guyeux, J.-F. Couchot, L. Marangio, and S. Galatolo, "A hardware and secure pseudorandom generator for constrained devices," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 8, pp. 3754–3765, 2018.

[17] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, 2017.

[18] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *USENIX Security*, 2019.

[19] A. G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson, "An architecture-independent instruction shuffler to protect against side-channel attacks," *ACM Transactions on Architecture and Code Optimations (TACO)*, 2012.

[20] R. Bernhard, P.-A. Moellic, and J.-M. Dutertre, "Impact of low-bitwidth quantization on the adversarial robustness for embedded neural networks," in *Proceedings of the International Conference on Cyberworlds (CW)*, 2019.

[21] A. Bhattacharjee, A. Moitra, and P. Panda, "Efficiency-driven hardware optimization for adversarially robust neural networks," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2021.

[22] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Annals of Mathematical Statistics*, vol. 29, pp. 610–611, 1958.

[23] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[24] J. Breier, D. Jap, X. Hou, S. Bhasin, and Y. Liu, "Sniff: Reverse engineering of neural networks with fault attacks," *IEEE Transactions on Reliability*, 2021.

[25] M. Brosch, M. Probst, and G. Sigl, "Counteract side-channel analysis of neural networks by shuffling," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2022.

[26] A. A. Bruen, M. A. Forcinito, and J. M. McQuillan, "Elliptic curve cryptography (ecc)," in *Cryptography, Information Theory, and Error-Correction: A Handbook for the 21st Century*, 2021, pp. 125–142.

[27] Z. Bu, Y.-X. Wang, S. Zha, and G. Karypis, *Automatic clipping: Differentially private deep learning made easier and stronger*, 2022. [Online]. Available: https://arxiv.org/abs/2206.07136.

[28] L. Cambier, A. Bhiwandiwalla, T. Gong, M. Nekuii, O. H. Elibol, and H. Tang, "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks," *arXiv preprint arXiv:2001.05674*, 2020.

[29] N. Carlini and D. Wagner, *Magnet and efficient defenses against adversarial attacks are not robust to adversarial examples*. [Online]. Available: {https://arxiv.org/abs/1711.08478}.

[30] N. Carlini and D. Wagner, "Adversarial examples are not easily detected: Bypassing ten detection methods," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017.

[31] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017.

[32] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, *Adversarial attacks and defences: A survey*, 2018. [Online]. Available: https://arxiv.org/abs/1810.00069.

[33] H. Chen *et al.*, "Anti-bandit neural architecture search for model defense," in *Proceedings of the European Conference on Computer Vision*, 2020, pp. 70–85.

[34] L. Chen, D. Moody, A. Regenscheid, and K. Randall, "Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters," National Institute of Standards and Technology, Tech. Rep., 2019.

[35] X. Chen and C.-J. Hsieh, "Stabilizing differentiable architecture search via perturbation-based regularization," in *Proceedings of the International conference on machine learning*, PMLR, 2020.

[36] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[37] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *Proceedings of the Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, 2019.

[38] Z. Chen, Y. Ma, and J. Jing, "Low-cost shuffling countermeasures against side-channel attacks for ntt-based post-quantum cryptography," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 1, 2023.

[39] Y. Cho *et al.*, "Artificial intelligence algorithm for detecting myocardial infarction using six-lead electrocardiography," *Scientific reports*, vol. 10, no. 1, 2020.

[40] W.-S. Choi, M. Tomei, J. R. S. Vicarte, P. K. Hanumolu, and R. Kumar, "Guaranteeing local differential privacy on ultra-low-power systems," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[41] T. Chouta *et al.*, "Side channel analysis on an embedded hardware fingerprint biometric comparator & low cost countermeasures," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014, pp. 1–6.

[42] J. Cohen, E. Rosenfeld, and Z. Kolter, "Certified adversarial robustness via randomized smoothing," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.

[43] J. C. Costa, T. Roxo, H. Proença, and P. R. Inácio, "How deep learning sees the world: A survey on adversarial attacks & defenses," 2023. [Online]. Available: https://arxiv.org/pdf/2305.10862.pdf.

[44] M. Costa, L. Esswood, O. Ohrimenko, F. Schuster, and S. Wagh, *The pyramid scheme: Oblivious RAM for trusted processors*, 2017. [Online]. Available: {https://arxiv.org/abs/1712.07882}.

[45] F. Croce and M. Hein, "Minimally distorted adversarial examples with a fast adaptive boundary attack," in *Proceedings of the International Conference on Machine Learning*, PMLR, 2020.

[46] F. Croce and M. Hein, "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks," in *Proceedings of the International Conference on Machine Learning*, PMLR, 2020.

[47] J.-L. Danger, S. Guilley, P. Hoogvorst, C. Murdica, and D. Naccache, "A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards," 4, vol. 3, Springer, 2013, pp. 241–265.

[48] H. Desai, M. Nardello, D. Brunelli, and B. Lucia, "Camaroptera: A long-range image sensor with local inference for remote sensing applications," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 3, 2022.

[49] S. N. Dhanuskodi and D. Holcomb, "Enabling microarchitectural randomization in serialized aes implementations to mitigate side channel susceptibility," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.

[50] A. Dhar, S. Sridhara, S. Shinde, S. Capkun, and R. Andri, *Empowering data centers for next generation trusted computing*, 2022. [Online]. Available: {https://arxiv.org/abs/2211.00306}.

[51] O. Dhifallah and Y. Lu, "On the inherent regularization effects of noise injection during training," in *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[52] *Division on ARM Cores*, https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/divide-and-conquer, 2022.

[53] J. Dofe, J. Frey, and Q. Yu, "Hardware security assurance in emerging iot applications," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016.

[54] H. M. Dolatabadi, S. Erfani, and C. Leckie, "Unleashing efficient adversarial training," in *Proceedings of the European Conference on Computer Vision*, Springer, 2022.

[55] L. Du *et al.*, "A reconfigurable streaming deep convolutional neural network accelerator for internet of things," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, 2018.

[56] A. Dubey, R. Cammarota, and A. Aysu, "Bomanet: Boolean masking of an entire neural network," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020.

[57] A. Dubey, R. Cammarota, and A. Aysu, "MaskedNet: The first hardware inference engine aiming power side-channel protection," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020.

[58] A. Dubey, R. Cammarota, A. Varna, R. Kumar, and A. Aysu, *Hardware-software co-design for side-channel protected neural network inference*, Cryptology ePrint Archive, Paper 2023/163, https://eprint.iacr.org/2023/163, 2023. [Online]. Available: https://eprint.iacr.org/2023/163.

[59] C. Dwork, "Differential privacy," in *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, Springer, 2006.

[60] A. Dziedzic and S. Krishnan, "Analysis of random perturbations for robust convolutional neural networks," *arXiv preprint arXiv:2002.03080*, 2020. [Online]. Available: https://arxiv.org/abs/2002.03080.

[61] G. K. Dziugaite, Z. Ghahramani, and D. M. Roy, *A study of the effect of jpg compression on adversarial images*, 2016. arXiv: 1608.00853. [Online]. Available: https://arxiv.org/abs/1608.00853.

[62] P. Echeverria and M. Lopez-Vallejo, "Fpga gaussian random number generator based on quintic hermite interpolation inversion," in *Proceedings of the 50th Midwest Symposium on Circuits and Systems*, 2007.

[63] H. Edrees, B. Cheung, M. Sandora, D. B. Nummey, and D. Stefan, "Hardware-optimized ziggurat algorithm for high-speed gaussian random number generators," in *ERSA*, 2009, pp. 254–260.

[64] P. Eustratiadis, *Wca-net*, https://github.com/peustr/WCA-Net, 2021.

[65] P. Eustratiadis, H. Gouk, D. Li, and T. Hospedales, "Weight-covariance alignment for adversarially robust neural networks," in *Proceedings of the 38th International Conference on Machine Learning*, PMLR, 2021.

[66] K. Eykholt *et al.*, "Robust physical-world attacks on deep learning models," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[67] A. Falcetta and M. Roveri, "Privacy-preserving deep learning with homomorphic encryption: An introduction," *IEEE Computational Intelligence Magazine*, vol. 17, no. 3, 2022.

[68] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[69] M. Feldhofer and T. Popp, "Power analysis resistant aes implementation for passive rfid tags," in *Austrochip 2008*, 2008, pp. 1–6.

[70] S. G. Finlayson, J. D. Bowers, J. Ito, J. L. Zittrain, A. L. Beam, and I. S. Kohane, "Adversarial attacks on medical machine learning," *Science*, vol. 363, no. 6433, pp. 1287–1289, 2019.

[71] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[72] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart, "Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[73] Y. Fu, Q. Yu, M. Li, V. Chandra, and Y. Lin, "Double-win quant: Aggressively winning robustness of quantized deep neural networks via random precision training and inference," in *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[74] Y. Fu, Y. Zhao, Q. Yu, C. Li, and Y. Lin, "2-in-1 accelerator: Enabling random precision switch for winning both adversarial robustness and efficiency," in *Proceedings of the 54th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[75] J. Galbally, "A new foe in biometrics: A narrative review of side-channel attacks," *Computers & Security*, vol. 96, 2020.

[76] M. Gallagher *et al.*, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *International Conference on Architecture Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019.

[77] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, "Ptolemy: Architecture support for robust deep learning," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020.

[78] K. Ganesan, V. Karyofyllis, J. Attai, A. Hamoda, and N. Enright Jerger, "Dinar: Enabling distribution agnostic noise injection in machine learning hardware," in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2023, pp. 38–46.

[79] J. Ge, N. Gao, C. Tu, J. Xiang, Z. Liu, and J. Yuan, "Combination of hardware and software: An efficient aes implementation resistant to side-channel attacks on all programmable soc," in *Computer Security*, Springer International Publishing, 2018.

[80] Q. Geng and P. Viswanath, "Optimal noise adding mechanisms for approximate differential privacy," *IEEE Transactions on Information Theory*, vol. 62, no. 2, pp. 952–969, 2015.

[81] H. Ghanbari, B. Khadem, and M. Jadidi, "Masking midori64 against correlation power analysis attack," *Biannual Journal Monadi for Cyberspace Security (AFTA)*, vol. 11, no. 1, 2022.

[82] B. Ghavami, S. Movi, Z. Fang, and L. Shannon, "Stealthy attack on algorithmic-protected dnns via smart bit flipping," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2022, pp. 1–7.

[83] B. Ghavami, M. Sadati, M. Shahidzadeh, Z. Fang, and L. Shannon, "Bdfa: A blind data adversarial bit-flip attack on deep neural networks," *arXiv preprint arXiv:2112.03477*, 2021.

[84] J. S. P Giraldo, S. Lauwereins, K. Badami, H. Van Hamme, and M. Verhelst, "18uw soc for near-microphone keyword spotting and speaker verification," in *2019 Symposium on VLSI Circuits*, 2019.

[85] T. Given-Wilson, N. Jafri, and A. Legay, "Combined software and hardware fault injection vulnerability detection," *Innovations in Systems and Software Engineering*, vol. 16, no. 2, pp. 101–120, 2020.

[86] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture," in *Proceedings of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2021.

[87] A. Golder, D. Das, J. Danial, S. Ghosh, S. Sen, and A. Raychowdhury, "Practical approaches toward deep-learning-based cross-device power side-channel attack," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[88] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," ser. STOC '87, 1987.

[89] M. Golec, S. S. Gill, R. Bahsoon, and O. Rana, "Biosec: A biometric authentication framework for secure and private communication among edge devices in iot and industry 4.0," *IEEE Consumer Electronics Magazine*, 2020.

[90] C. Gongye, Y. Fei, and T. Wahl, "Reverse-engineering deep neural networks using floating-point timing side-channels," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[91] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[92] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, 2014. DOI: 10.48550/ARXIV.1412.6572. [Online]. Available: https://arxiv.org/abs/1412.6572.

[93] M. Greenacre, P. J. Groenen, T. Hastie, A. I. d'Enza, A. Markos, and E. Tuzhilina, "Principal component analysis," *Nature Reviews Methods Primers*, vol. 2, no. 1, p. 100, 2022.

[94] S. Gu and L. Rigazio, *Towards deep neural network architectures robust to adversarial examples*, 2014. [Online]. Available: https://arxiv.org/abs/1412.5068.

[95] J. Guérin, O. Gibaru, S. Thiery, and E. Nyiri, "Cnn features are also great at unsupervised classification," *arXiv preprint arXiv:1707.01700*, 2017.

[96] A. Guesmi *et al.*, "Defensive approximation: Securing cnns using approximate computing," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[97] C. Guo, M. Rana, M. Cisse, and L. van der Maaten, "Countering adversarial images using input transformations," in *Proceedings of the International Conference on Learning Representations*, 2018.

[98] M. Guo, Y. Yang, R. Xu, Z. Liu, and D. Lin, "When NAS meets robustness: In search of robust architectures against adversarial attacks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 631–640.

[99] R. Guo *et al.*, "A 5.1pj/neuron 127.3us/inference rnn-based speech recognition processor using 16 computing-in-memory sram macros in 65nm cmos," in *Proceedings of the International Symposium on VLSI Circuits*, 2019.

[100] S. Gupta, P. Dube, and A. Verma, "Improving the affordability of robustness training for dnns," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 2020.

[101] R. Gutierrez, V. Torres, and J. Valls, "Hardware architecture of a gaussian noise generator based on the inversion method," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 8, 2012.

[102] S. Ha and S. Choi, "Convolutional neural networks for human activity recognition using multiple accelerometer and gyroscope sensors," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016.

[103] V. Hadžić and R. Bloem, "Cocoalma: A versatile masking verifier," in *Conference on Formal Methods in Computer-aided Design – FMCAD 2021*, 2021.

[104] M. Hassanalieragh *et al.*, "Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges," in *2015 IEEE International Conference on Services Computing*, 2015.

[105] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, "A hybrid approach to offloading mobile image classification," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[106] T. K. Hazra and S. Bhattacharyya, "Image encryption by blockwise pixel shuffling using modified fisher yates shuffle and pseudorandom permutations," in *Proceedings of the 7th Annual IEEE Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2016.

[107] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015.

[108] S. He, W. Wu, Y. Li, L. Zhou, L. Fang, and Z. Liu, "Recovering the weights of convolutional neural network via chosen pixel horizontal power analysis," in *Wireless Algorithms, Systems, and Applications: 17th International Conference, WASA 2022, Dalian, China, November 24–26, 2022, Proceedings, Part II*, Springer, 2022, pp. 93–104.

[109] Z. He, A. S. Rakin, and D. Fan, "Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 588–597.

[110] M. Hicks, *Thumbulator: Cycle accurate ARMv6-m instruction set simulator*. https://bit.ly/2RJX36A, 2016.

[111] M. Hicks, "Clank: Architectural Support for Intermittent Computation," in *ISCA*, 2017.

[112] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[113] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[114] K. T. Hokuto Hirano Akinori Minagi, "Universal adversarial attacks on deep neural networks for medical image classification," *BMC Medical Imaging*, vol. 21, no. 9, 2021.

[115] S. Hong *et al.*, "Security analysis of deep neural networks operating in the presence of cache side-channel attacks," *CoRR*, vol. abs/1810.03487, 2018. arXiv: 1810.03487. [Online]. Available: http://arxiv.org/abs/1810.03487.

[116] S. Hori, T. Shouo, K. Gyohten, H. Ohki, T. Takami, and N. Sato, "Arrhythmia detection based on patient-specific normal ecgs using deep learning," in *2020 Computing in Cardiology*, 2020.

[117] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "Guardnn: Secure dnn accelerator for privacy-preserving deep learning," *arXiv preprint arXiv:2008.11632*, 2020.

[118] W. Hua, Y. Zhang, C. Guo, Z. Zhang, and G. E. Suh, "Bullettrain: Accelerating robust neural network training via boundary example mining," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[119] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[120] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[121] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[122] U. Hwang, J. Park, H. Jang, S. Yoon, and N. I. Cho, "Puvae: A variational autoencoder to purify adversarial examples," *IEEE Access*, vol. 7, pp. 126 582–126 593, 2019.

[123] "Ieee standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985. DOI: 10.1109/IEEESTD.1985.82928.

[124] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the International conference on machine learning*, 2015, pp. 448–456.

[125] S. Islam, I. Alouani, and K. N. Khasawneh, "Lower voltage for higher security: Using voltage overscaling to secure deep neural networks," in *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[126] A. Jain *et al.*, "Overview and importance of data quality for machine learning tasks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[127] M. Javaheripi and F. Koushanfar, "Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[128] A. Jeddi, M. J. Shafiee, M. Karg, C. Scharfenberger, and A. Wong, "Learn2perturb: An end-to-end feature perturbation learning to improve adversarial robustness," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[129] X. Jia, X. Wei, X. Cao, and H. Foroosh, "Comdefend: An efficient image compression model to defend adversarial examples," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 6084–6092.

[130] K. Jiang, B. Zhao, W. Shan, L. Wang, and J. Liu, "Profiling attack on modular multiplication of elliptic curve cryptography," in *International Conference on Computational Intelligence and Security (CIS)*, 2016.

[131] J. Jin, E. McMurtry, B. I. Rubinstein, and O. Ohrimenko, "Are we there yet? timing and floating-point attacks on differential privacy systems," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 473–488.

[132] I. T. Jolliffe and J. Cadima, "Principal component analysis: A review and recent developments," *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, 2016.

[133] R. Joud, P.-A. Moëllic, S. Pontié, and J.-B. Rigaud, "A practical introduction to side-channel extraction of deep neural network parameters," in *Proceedings of the 21st International Conference on Smart Card Research and Advanced Applications (CARDIS)*, 2023.

[134] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Computer Architecture News*, vol. 45, no. 2, 1–12, 2017.

[135] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[136] A. Karpur, D. Lahav, J. Matheny, J. Alstott, and S. Nevo, *Securing artificial intelligence model weights: Interim report*. [Online]. Available: {https://www.rand.org/content/dam/rand/pubs/working_papers/WRA2800/WRA2849-1/RAND_WRA2849-1.pdf}.

[137] L. Kaufman, "Partitioning around medoids," *Finding groups in data*, vol. 344, pp. 68–125, 1990.

[138] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[139] F. Khalid *et al.*, "QuSecNets: Quantization-based defense mechanism for securing deep neural network against adversarial attacks," *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019.

[140] H. "Kim, S. Hong, and J. Lim, "A fast and provably secure higher-order masking of aes s-box," in *Cryptographic Hardware and Embedded Systems*, 2011.

[141] H. Kim, *Torchattacks: A pytorch repository for adversarial attacks*, 2020. [Online]. Available: https://arxiv.org/abs/2010.01950.

[142] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms," in *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2018.

[143] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[144] O. Kuchaiev *et al.*, "Mixed-precision training for nlp and speech recognition with openseq2seq," *arXiv preprint arXiv:1805.10387*, 2018.

[145] S. Kundu, J.-P. D'Anvers, M. Van Beirendonck, A. Karmakar, and I. Verbauwhede, "Higher-order masked saber," in *Proceedings of the 13th International Conference on Security and Cryptography for Networks (SCN)*, 2022, pp. 93–116.

[146] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[147] V. Këpuska and G. Bohouta, "Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home)," in *Proceedings of the IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 2018.

[148] J. Lagasse, C. Bartoli, and W. Burleson, "Combining clock and voltage noise countermeasures against power side-channel analysis," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2019.

[149] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana, "Certified robustness to adversarial examples with differential privacy," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 656–672.

[150] D.-U. Lee, J. D. Villasenor, W. Luk, and P. H. W. Leong, "A hardware gaussian noise generator using the box-muller method and its error analysis," *IEEE transactions on computers*, vol. 55, no. 6, pp. 659–671, 2006.

[151] D.-U. Lee, R. C. Cheung, J. D. Villasenor, and W. Luk, "Inversion-based hardware gaussian random number generator: A case study of function evaluation via hierarchical segmentation," in *Proceedings of the IEEE International Conference on Field Programmable Technology*, 2006.

[152] J. Lee and C. Clifton, "How much is enough? choosing ε for differential privacy," in *Proceedings of the 14th International Conference on Information Security (ISC)*, 2011.

[153] J. Lee and D.-G. Han, "Security analysis on dummy based side-channel countermeasures—case study: Aes with dummy and shuffling," *Applied Soft Computing*, vol. 93, p. 106 352, 2020.

[154] J.-W. Lee *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, 2022.

[155] H. Li, J. Li, X. Guan, B. Liang, Y. Lai, and X. Luo, "Research on overfitting of deep learning," in *Proceedings of the 15th International Conference on Computational Intelligence and Security (CIS)*, 2019, pp. 78–81.

[156] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance analysis of gpu-based convolutional neural networks," in *2016 45th International conference on parallel processing (ICPP)*, IEEE, 2016, pp. 67–76.

[157] Y. Li, P. Zhao, X. Lin, B. Kailkhura, and R. Goldh, *Less is more: Data pruning for faster adversarial training*, 2023. [Online]. Available: {https://arxiv.org/abs/2302.12366}.

[158] J. Lin, C. Gan, and S. Han, "Defensive quantization: When efficiency meets robustness," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[159] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018.

[160] X. Liu, M. Cheng, H. Zhang, and C.-J. Hsieh, "Towards robust neural networks via random self-ensemble," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 369–385.

[161] Y. Liu, D. Dachman-Soled, and A. Srivastava, "Mitigating reverse engineering attacks on deep neural networks," in *IEEE Computer Society Annual Symposium on VLSI*, 2019.

[162] J. Lu, T. Issaranon, and D. Forsyth, "Safetynet: Detecting and rejecting adversarial examples robustly," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 446–454.

[163] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[164] S. A. Magid, F. Petrini, and B. Dezfouli, "Image classification on iot edge devices: Profiling and modeling," *Cluster Computing*, vol. 23, no. 2, pp. 1025–1043, 2020.

[165] S. Maji, U. Banerjee, and A. P. Chandrakasan, "Leaky nets: Recovering embedded neural network models and inputs through simple power and timing side-channels—attacks and defenses," *IEEE Internet of Things Journal*, vol. 8, no. 15, 2021.

[166] S. Maji, U. Banerjee, S. H. Fuller, and A. P. Chandrakasan, "A threshold implementation-based neural network accelerator with power and electromagnetic side-channel countermeasures," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, 2023.

[167] S. Majumdar, M. H. Samavatian, K. Barber, and R. Teodorescu, "Using undervolting as an on-device defense against adversarial machine learning attacks," in *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, 2021.

[168] J. S. Malik and A. Hemani, "Gaussian random number generation: A survey on hardware architectures," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, 2016.

[169] P. M. Mammen, *Federated learning: Opportunities and challenges*, 2021. [Online]. Available: https://arxiv.org/abs/2101.05428.

[170] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag New York, Inc., 2007.

[171] E. B. S. Martin, H. Shirazi, and I. Ray, "Poster: Towards a dataset for the discrimination between warranted and unwarranted emails," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[172] D. Mautz, C. Plant, and C. Böhm, "Deep embedded cluster tree," in *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2019, pp. 1258–1263.

[173] M. Medwed, F.-X. Standaert, J. Großschädl, and F. Regazzoni, "Fresh re-keying: Security against side-channel and fault attacks for low-cost devices.," *AFRICACRYPT*, vol. 6055, pp. 279–296, 2010.

[174] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *Proceedings of the IEEE symposium on security and privacy (SP)*, 2019.

[175] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, "Mixed precision training with 8-bit floating point," *arXiv preprint arXiv:1905.12334*, 2019.

[176] F. Meneghello, M. Calore, D. Zucchetto, M. Polese, and A. Zanella, "IoT: Internet of threats? a survey of practical security vulnerabilities in real IoT devices," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8182–8201, 2019.

[177] D. Meng and H. Chen, "Magnet: A two-pronged defense against adversarial examples," in *Proceedings of the Conference on Computer and Communications Security (SIGSAC*, 2017, pp. 135–147.

[178] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2019.

[179] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, *On detecting adversarial perturbations*, 2017. DOI: 10.48550/ARXIV.1702.04267. [Online]. Available: https://arxiv.org/abs/1702.04267.

[180] P. Micikevicius *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[181] P. Micikevicius *et al.*, *Fp8 formats for deep learning*, 2022. [Online]. Available: {https://arxiv.org/abs/2209.05433}.

[182] P. Micikevicius *et al.*, "Fp8 formats for deep learning," *arXiv preprint arXiv:2209.05433*, 2022.

[183] *Microcontroller Advances from STMicroelectronics Extend Performance Leadership for Smarter Technology Everywhere*, https://bit.ly/2PofKuy, ST Microelectronics, 2013.

[184] F. Mireshghallah, M. Taram, A. Jalali, A. T. T. Elthakeb, D. Tullsen, and H. Esmaeilzadeh, "Not all features are equal: Discovering essential features for preserving prediction privacy," in *Proceedings of the Web Conference 2021*, 2021.

[185] F. Mireshghallah, M. Taram, P. Ramrakhyani, A. Jalali, D. Tullsen, and H. Esmaeilzadeh, "Shredder: Learning noise distributions to protect inference privacy," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[186] I. Mironov, "On significance of the least significant bits for differential privacy," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[187] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for IoT big data and streaming analytics: A survey," *IEEE Communications Surveys and Tutorials*, 2018.

[188] A. Moradi, O. Mischke, and C. Paar, "Practical evaluation of dpa countermeasures on reconfigurable hardware," in *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2011.

[189] K. Mpalane, N. Gasela, B. M. Esiefarienrhe, and H. D. Tsague, "Vulnerability of advanced encryption standard algorithm to differential power analysis attacks implemented on ATmega-128 microcontroller," in *2016 Third International Conference on Artificial Intelligence and Pattern Recognition (AIPR)*, 2016.

[190] A. Muhammad and S.-H. Bae, "A survey on efficient methods for adversarial robustness," *IEEE Access*, vol. 10, 2022.

[191] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

[192] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine learning at the network edge: A survey," *ACM Computing Surveys*, vol. 54, no. 8, 2021.

[193] J. Myers, A. Savanth, R. Gaddh, D. Howard, P. Prabhat, and D. Flynn, "An 80nw retention 11.7pj/cycle active subthreshold arm cortex-m0+ subsystem in 65nm cmos for wsn applications," *ISSCC*, 2015.

[194] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? Adaptive rounding for post-training quantization," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 119, 2020.

[195] Y. Nahshan *et al.*, "Loss aware post-training quantization," *Machine Learning*, vol. 110, no. 11-12, pp. 3245–3262, 2021.

[196] S. Naumov, G. Yaroslavtsev, and D. Avdiukhin, "Objective-based hierarchical clustering of deep embedding vectors," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 9055–9063.

[197] C. Nayak, *New privacy-protected facebook data for independent research on social media's impact on democracy*, https://research.facebook.com/blog/2020/2/new-privacy-protected-facebook-data-for-independent-research-on-social-medias-impact-on-democracy/, accessed on 2022-06-11, 2020.

[198] H. Noh, T. You, J. Mun, and B. Han, "Regularizing deep neural networks by noise: Its interpretation and optimization," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[199] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi, "8-bit numerical formats for deep neural networks," *arXiv preprint arXiv:2206.02915*, 2022.

[200] Y. Nozaki and M. Yoshikawa, "Shuffling countermeasure against power side-channel attack for mlp with software implementation," in *Proceedings of the International Conference on Electronics and Communication Engineering (ICECE)*, 2021.

[201] *Nvidia Ada Lovelace GPU Architecture*, 2023. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[202] *Nvidia Ampere GPU architecture whitepaper*, 2023. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[203] *NVIDIA Hopper Architecture In-Depth*, 2023. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth.

[204] *Nvidia Tensor Core Datasheet*, 2023. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.

[205] C. O'Flynn and Z. D. Chen, "ChipWhisperer: An open-source platform for hardware embedded security research," in *Constructive Side-Channel Analysis and Secure Design*, 2014, pp. 243–260.

[206] S. Pallavi, J. D. Mallapur, and K. Y. Bendigeri, "Remote sensing and controlling of greenhouse agriculture parameters based on iot," in *2017 International Conference on Big Data, IoT and Data Science (BID)*, 2017.

[207] V. R. Pamula, X. Sun, S. Kim, F. u. Rahman, B. Zhang, and V. S. Sathe, "An all-digital true-random-number generator with integrated de-correlation and bias correction at 3.2-to-86 mb/s, 2.58 pj/bit in 65-nm cmos," in *IEEE Symposium on VLSI Circuits*, 2018.

[208] P. Panda, "QUANOS adversarial noise sensitivity driven hybrid quantization of neural networks," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED*, 2020.

[209] T. Pang, X. Yang, Y. Dong, H. Su, and J. Zhu, "Bag of tricks for adversarial training," *arXiv preprint arXiv:2010.00467*, 2020.

[210] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 582–597. DOI: 10.1109/SP.2016.41.

[211] B. Park, R. Hwang, D. Yoon, Y. Choi, and M. Rhu, "Diva: An accelerator for differentially private machine learning," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[212] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[213] S. Patranabis, D. B. Roy, P. K. Vadnala, D. Mukhopadhyay, and S. Ghosh, "Shuffling across rounds: A lightweight strategy to counter side-channel attacks," in *IEEE International Conference on Computer Design (ICCD)*, 2016.

[214] K. Pier, *MSP Code Protection Features*, https://www.ti.com/lit/an/slaa685/slaa685.pdf?ts=1595266414590, TI, 2015.

[215] R. Pinot, L. Meunier, F. Yger, C. Gouy-Pailler, Y. Chevaleyre, and J. Atif, "On the robustness of randomized classifiers to adversarial examples," *arXiv preprint arXiv:2102.10875*, 2021.

[216] R. Pinot *et al.*, "Theoretical evidence for adversarial robustness through randomization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[217] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, 2016.

[218] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[219] D. Roy, I. Chakraborty, T. Ibrayev, and K. Roy, "On the intrinsic robustness of nvm crossbars against adversarial attacks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[220] D. Roy, C. Tao, I. Chakraborty, and K. Roy, *On the noise stability and robustness of adversarially trained networks on nvm crossbars*, 2021. [Online]. Available: https://arxiv.org/abs/2109.09060.

[221] S. Saeed, M. S. Umar, M. A. Ali, and M. Ahmad, "A gray-scale image encryption using fisher-yates chaotic shuffling in wavelet domain," in *Proceedings of the IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*, 2014.

[222] R. Sahay, R. Mahfuz, and A. E. Gamal, *A computationally efficient method for defending adversarial deep learning attacks*, 2019. [Online]. Available: https://arxiv.org/abs/1906.05599.

[223] P. Samangouei, M. Kabkab, and R. Chellappa, "Defense-gan: Protecting classifiers against adversarial attacks using generative models," *arXiv preprint arXiv:1805.06605*, 2018.

[224] M. H. Samavatian, S. Majumdar, K. Barber, and R. Teodorescu, *Dnnshield: Dynamic randomized model sparsification, a defense against adversarial machine learning*, 2022. [Online]. Available: {https://arxiv.org/abs/2208.00498}.

[225] S. S. I. Samuel, "A review of connectivity challenges in iot-smart home," in *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, 2016.

[226] M. Schulte and E. Swartzlander, "Hardware designs for exactly rounded elementary functions," *IEEE Transactions on Computers*, vol. 43, no. 8, 1994.

[227] C. H. Seng, R. Demirli, L. Khuon, and D. Bolger, "Seizure detection in eeg signals using support vector machines," in *2012 38th Annual Northeast Bioengineering Conference (NEBEC)*, 2012.

[228] A. Shafahi *et al.*, "Adversarial training for free!" *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[229] *Silicon Gecko M0+ EFM32TG11*, https://www.silabs.com/documents/public/reference-manuals/efm32tg11-rm.pdf(pages1062-1078), Silicon Gecko, 2020, pp. 1062 –1078.

[230] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2014. [Online]. Available: {https://arxiv.org/abs/1409.1556}.

[231] C. Sitawarin and D. Wagner, *Defending against adversarial examples with k-nearest neighbor*, 2019. DOI: 10.48550/ARXIV.1906.09525. [Online]. Available: https://arxiv.org/abs/1906.09525.

[232] Y. Song, T. Kim, S. Nowozin, S. Ermon, and N. Kushman, "Pixeldefend: Leveraging generative models to understand and defend against adversarial examples," *arXiv preprint arXiv:1710.10766*, 2017.

[233] *STM32L0 ARM M0+ CPUs*, https://www.st.com/en/microcontrollers-microprocessors/stm32l0-series.html, ST Micro.

[234] B. Sunar, "True random number generators for cryptography," in *Cryptographic Engineering*. 2009, pp. 55–73.

[235] N. H. Tandel, H. B. Prajapati, and V. K. Dabhi, "Voice recognition and voice comparison using machine learning techniques: A survey," in *Proceedings of the IEEE 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020.

[236] M. Tayel, G. Dawood, and H. Shawky, "Block cipher s-box modification based on fisher-yates shuffle and ikeda map," in *Proceedings of the IEEE 18th International Conference on Communication Technology (ICCT)*, 2018.

[237] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[238] *Tesla Volta V100 Datasheet*, 2023. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

[239] V. Thakkar, S. Tewary, and C. Chakraborty, "Batch normalization in convolutional neural networks — a comparative study with cifar-10 data," in *2018 Fifth International Conference on Emerging Applications of Information Technology (EAIT)*, 2018.

[240] K. Thakur, M. L. Ali, M. A. Obaidat, and A. Kamruzzaman, "A systematic review on deep-learning-based phishing email detection," *Electronics*, vol. 12, no. 21, 2023.

[241] T. Titcombe, A. J. Hall, P. Papadopoulos, and D. Romanini, "Practical defences against model inversion attacks for split neural networks," in *Proceedings of the Workshop on Distributed and Private Machine Learning (DPML) co-located with ICLR*, 2021.

[242] S. Venkataramani *et al.*, "Rapid: Ai accelerator for ultra-low precision training and inference," in *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[243] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *(ASIACRYPT)*, 2012.

[244] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[245] X. Wang, R. Hou, Y. Zhu, J. Zhang, and D. Meng, "Npufort: A secure architecture of dnn accelerator against model inversion attack," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 190–196.

[246] X. Wang, B. Zhao, R. Hou, A. Awad, Z. Tian, and D. Meng, "Nasguard: A novel accelerator architecture for robust neural architecture search (nas) networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.

[247] X. Wang *et al.*, "Dnnguard: An elastic heterogeneous dnn accelerator architecture against adversarial attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[248] Y. Wang and Y. Ha, "An area-efficient shuffling scheme for AES implementation on FPGA," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013.

[249] Y. Wang, D. Zou, J. Yi, J. Bailey, X. Ma, and Q. Gu, "Improving adversarial robustness requires revisiting misclassified examples," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[250] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.

[251] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. Xu, "I know what you see: Power side-channel attack on convolutional neural network accelerators," *Computer Security Applications Conference*, 2018.

[252] D. Williams-King *et al.*, "Shuffler: Fast and deployable continuous code re-randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[253] E. Wong, L. Rice, and J. Z. Kolter, *Fast is better than free: Revisiting adversarial training*, 2020. [Online]. Available: {https://arxiv.org/abs/2001.03994}.

[254] M. Wortsman, T. Dettmers, L. Zettlemoyer, A. Morcos, A. Farhadi, and L. Schmidt, "Stable and low-precision training for large-scale vision-language models," *arXiv preprint arXiv:2304.13013*, 2023.

[255] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2019.

[256] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *USENIX Security Symposium*, 2020.

[257] D. Yang, P. J. Nair, and M. Lis, "Huffduff: Stealing pruned dnns from sparse accelerators," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[258] Y. Yang, G. Zhang, D. Katabi, and Z. Xu, "Me-net: Towards effective adversarial robustness with matrix estimation," *arXiv preprint arXiv:1905.11971*, 2019.

[259] A. Yousefpour *et al.*, "Opacus: User-friendly differential privacy library in PyTorch," *arXiv preprint arXiv:2109.12298*, 2021.

[260] V. Zantedeschi, M.-I. Nicolae, and A. Rawat, "Efficient defenses against adversarial attacks," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017.

[261] F. Zhan, "Hand gesture recognition with convolution neural networks," in *IEEE International Conference on Information Reuse and Integration for Data Science (IRI)*, 2019.

[262] G. Zhang, P. H. W. Leong, D.-U. Lee, J. D. Villasenor, R. C. Cheung, and W. Luk, "Ziggurat-based hardware gaussian random number generator," in *Proceedings of the International Conference on Field Programmable Logic and Applications, 2005.*, IEEE, 2005, pp. 275–280.

[263] H. Zhang, Y. Yu, J. Jiao, E. Xing, L. El Ghaoui, and M. Jordan, "Theoretically principled trade-off between robustness and accuracy," in *Proceedings of the International conference on machine learning*, PMLR, 2019.

[264] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. abs/1711.07128, 2017. arXiv: 1711.07128. [Online]. Available: http://arxiv.org/abs/1711.07128.

[265] P. Zhao, S. Wang, C. Gongye, Y. Wang, Y. Fei, and X. Lin, "Fault sneaking attack: A stealthy framework for misleading deep neural networks," in *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[266] R. Zhao, B. Vogel, and T. Ahmed, "Adaptive loss scaling for mixed precision training," *arXiv preprint arXiv:1910.12385*, 2019.

[267] R. Zhao, B. Vogel, T. Ahmed, and W. Luk, "Reducing underflow in mixed precision training by gradient scaling," in *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, 2021, pp. 2922–2928.

[268] E. Zheltonozhskii, C. Baskin, Y. Nemcovsky, B. Chmiel, A. Mendelson, and A. M. Bronstein, "Colored noise injection for training adversarially robust neural networks," *arXiv preprint arXiv:2003.02188*, 2020.

[269] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Towards effective low-bitwidth convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7920–7928.

[270] M. Znalezniak, P. Rola, P. Kaszuba, J. Tabor, and M. Śmieja, "Contrastive hierarchical clustering," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2023, pp. 627–643.