Secure Accelerator Design for Deep Neural Networks

by

Lei Zhao

Master of Computer Science, Northwestern Polytechnical University, China, 2014

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2022

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Lei Zhao

It was defended on

April 8th 2022

and approved by

Secure Accelerator Design for Deep Neural Networks

Lei Zhao, PhD

University of Pittsburgh, 2022

Deep neural networks (DNNs) have recently gained popularity in a wide range of modern application domains due to its superior inference accuracy. With growing problem size and complexity, modern DNNs, e.g., CNNs (convolutional neural networks), contain a large number of weights, which require tremendous efforts not only to prepare representative training data but also to train the network. There is an increasing demand to protect the DNN weights, an emerging intellectual property (IP) in the DNN field. This thesis proposes a line of solutions for protecting the DNN weights deployed on domain specific accelerators.

Firstly, I propose `AEP`, a DNN weights protection scheme for accelerators based on conventional CMOS based technologies. Because of the extremely high memory bandwidth demand in DNN accelerators, conventional encryption based approaches, which require the integration of expensive encryption engines, pose significant overheads on the execution latency and energy consumption. Instead, `AEP` enables effective IP protection by utilizing fingerprints generated from hardware characteristics to eliminate the need of encryption. Adopting such hardware fingerprints achieves high inference accuracy only on the authorized device, while unauthorized devices can not produce any useful results from the same set of weights.

Secondly, as the size of DNNs keeps increasing rapidly, the large number of intermediate results (i.e, the outputs from the previous layer and the inputs to the current layer) can not be held on-chip. These intermediate results also contain sensitive information about the DNN itself. In this part, I propose `SCA` which can securely off-load data dynamically generated inside the accelerator chip to off-chip memories. `SCA` is a full DNN protection scheme that protects both the DNN weights and the intermediate results, and supports both training and inference on CMOS based accelerators.

Thirdly, ReRAM based accelerators introduce new challenges to DNN IP protection due to their crossbar structure and non-volatility. ReRAM's non-volatility retains data even after the system is powered off, making the stored DNN weights vulnerable to attacks by simply reading out the ReRAM content. Because the crossbar structure can only compute on cleartext data, encrypting the ReRAM content is no longer a feasible solution in this scenario. To solve these issues, I propose `SRA`, a novel non-encryption base protection method that still maintains ReRAM's in-memory computing capability.

Lastly, although `SRA` provides security guarantees, the weights are represented in stochastic computing (SC) bit stream format, which induces a large storage overhead. However, conventional DNN model compression methods, such as pruning and quantization, are not applicable ReRAM based PIM accelerators. In this part, I propose `BFlip` — a novel DNN model compression scheme — to share crossbars among multiple bit matrices. `BFlip` not only reduces storage overhead but also improves performance and energy efficiency.

Keywords: neural network, security, accelerator, ReRAM, in-situ.

Table of Contents

List of Tables

List of Figures

Preface

I would like to thank all the people that I have met and worked with during my Ph.D study. First and foremost, I would like to thank my advisor, Professor Youtao Zhang, for his endless support. During the Ph.D program, he has put tremendous amount of effort to guide me through each step to explore and research. I am also grateful for the helps from Professor Jun Yang and Professor Xulong Tang for their comments and valuable advice on my reserach and presentations in our periodic lab-wise meeting. They taught me lots of things about doing research, and constantly provided insightful discussions and feedbacks on on my research projects and conference presentations.

Sincere thanks to all the committee members: Professor Youtao Zhang, Professor Jun Yang, Professor Adriana Kovashka and Professor Xulong Tang, for their time, efforts and valuable inputs into both the thesis proposal and final dissertation. Without their great helps, this dissertation would not be possible.

Thanks to all the colleagues in Professor Zhang's lab. The interesting and enjoyable discussions helped a lot in my research works. I had a great time with them for the past few years both inside and outside research.

Finally, I take this opportunity to express my gratitute to my family for their love, unfailing encouragement and support.

1.0 INTRODUCTION

Deep Neural Networks (DNNs) are machine learning approaches that, by exploiting the rapidly increasing computation power of modern computers, achieve superior inference accuracy improvements over traditional machine learning algorithms. Therefore, DNNs are gaining popularity in a wide range of modern application domains, e.g., computer vision [21] and speech recognition [32].

A typical DNN is composed of its layer structure as well as the massive amount of weights in each layer. While the layer structure is usually well known (i.e., the layers of a DNN are from a small set of common layer types), the weight values are more problem dependent and require not only a large number of computing iterations to train but also the efforts to prepare the effective training datasets which may include sensitive and/or proprietary data. Therefore, the DNN (especially the trained weights) is an intellectual property (IP) that needs to be protected. Leaking such information will result in huge losses. On the one hand, adversaries may abuse the DNN without getting permission. For example, the object tracking and recognition DNN of an autonomous driving car may be pirated, which speeds up the development of new systems from the competitors. On the other hand, adversaries may perform security attacks on the weights, such as changing the output of a DNN by adversarial examples.

Within each layer, the computations on the weights are highly parallelizable. The intrinsic memory and computation intensity of DNNs has driven the development of hardware accelerators for high performance and energy efficiency. For example, the DianNao accelerator [14] achieves 118$\times$ performance speedup and 21$\times$ energy reduction over an SIMD CPU. However, the massive amount of memory access remains a major bottleneck for hardware DNN accelerators, as concluded in [14]. This poses a challenge in using the conventional encryption based methods to protect DNNs that are deployed on accelerators. There is a huge demand for developing new efficient IP protection schemes for DNN accelerators.

1.1 Problem Statement

For the DNNs that are deployed on the client side, it is challenging to design effective protection solutions. A naive solution to protect DNNs is to use encryption. To convoy the weights to the accelerator without being exposed to the third party during the process, we may adopt the XOM design [50] and the privacy enhancement [82]. It works briefly as follows. We first upgrade the accelerator with a small TCB (trusted computing base) that contains a crypto engine and PKI (public key infrastructure) support, i.e., the private key is kept secret while the public key is released to the end users and servers. The server encrypts the weights using a session key and then encrypts the session key with the public key of the accelerator. Thus, only the target accelerator can decrypt the session key and then decrypt the encrypted weights before computation.

Unfortunately, this simple solution faces severe scalability issues. Given that DNNs are increasingly adopted to accomplish challenging tasks in various application domains, their sizes, i.e., the number of weights in the network, grow rapidly. For example, Le *et al.* [43] created a DNN with 1 billion weight parameters, Catanzaro *et al.* [18] used a DNN that has 11 billion parameters. More recently, Ni *et al.* [57] built a DNN that contains over 15 billion parameters, which requires more than 30GB if using 16-bit fixed point. It consists of two convolutional layers and one classifier layer, which have 3GB, 18GB, and 12GB weights, respectively.

A DNN accelerator, due to its limited on-chip memory, needs to load and decrypt a large number of weights when starting a new layer. Adopting High Bandwidth Memory (HBM) improves the maximal bandwidth but not the decryption latency. For example, without encrypting and decrypting the weights, the second layer of the above DNN [57] finishes in 504μs using DianNao. However, even we optimistically assume that the weights of the third layer (i.e., 12GB) can be loaded in parallel with the execution of the second layer, decrypting the weights with a fully-unrolled, pipelined AES implementation needs 349ms [3]. Here we optimistically assume the AES decryption engine consists of multiple copies such that it can match the peak HBM bandwidth, i.e., 256GB/s. The weight decryption can easily become the latency bottleneck of the whole system.

In addition, the encryption based scheme is not applicable to process-in-memory (PIM) accelerators. For example, ReRAM based accelerators store the DNN weights as the resistance of ReRAM cells which are organized in a crossbar structure, while the inputs are fed into the crossbar in the form of voltage levels. The matrix vector multiplication (MVM) is performed based on Kirchoff's Law, i.e., the current flowing out represents the MVM results. This computation paradigm can only compute on cleartext data.

Therefore, the encryption based scheme is less preferred and we need to find another more efficient way to provide effective DNN weights protection.

1.2 Research Overview

Our research aims to achieve the following goals: (i) protect the embedded DNNs with little or no computation overhead; (ii) guarantee the security of the deployed DNN inside the accelerators; (iii) support accelerators based on a variety of technologies.

To achieve the above goals, in this thesis we propose a series of hardware based DNN protection schemes on accelerators. Firstly, we propose to utilize the intrinsic hardware characteristics to protect the embedded DNNs, thus avoiding the expensive encryption and decryption operations. Secondly, our proposed schemes hide the actual weights that are used in computation after loading onto the accelerators. Even if the host CPU and/or off-chip memory are compromised, the security of the DNNs can still be guaranteed during execution. Lastly, we extend our schemes onto non-volatile in-situ accelerators. Since in-situ computing can only operate on cleartext data, we propose a novel shuffle based protection scheme to maintain the in-situ computing capabilities. To tackle the storage and computing overheads in the shuffle based protection scheme, we also propose novel model compression schemes for ReRAM accelerators.

1.2.1 Protect Weights with Hardware Fingerprints

DNNs are usually over-parameterized, which makes them tolerant to errors in the computing data as well as the computation itself. At the same time, DNNs usually find local minima on the loss surface as their solution. Given these two characteristics, we observe that if we first train the DNN with stuck errors at some fixed positions in the DNN weights (i.e., making the corresponding bits of the data stuck at 0 or 1 during the entire training process), the trained DNN can still produce high accurate predictions on classification tasks. However, if we manually flip the previously stuck bits in the trained weights (i.e., change the bit to 1 if it was stuck at 0 during training, change the bit to 0 if it was stuck at 1 during training), the modified DNN will significantly reduce its prediction accuracy, making the DNN unusable.

From the above observation, we utilize this phenomenon to restrict the DNN on a specific hardware [83]. Specifically, we extract the error distribution in the memory of a specific device. It has been reported that the DRAM timing induced errors follow a fixed spatial distribution, but change across different individual devices [2]. This error distribution can be used as the unique fingerprint for this device. We then train the DNN with these errors statically present in the weights to get a well-functioning DNN. Before deploying the DNN on the device, we manually flip the previous error bits to make the DNN a malfunctioning one. During run-time, the device uses the same DRAM timing to generate errors, thus automatically flipping the errors bits back to make the DNN well-functioning again.

The well-functioning weights only exist during run-time after they are loaded into the DRAM. This makes the DNN weights stored outside the DRAM and accelerator useless to the adversaries, thus effectively protecting the DNN IP. At the same time, the reduced DRAM timing can also bring performance and energy improvements.

1.2.2 Protect Weights and Intermediate Results with Hardware Fingerprints

Many DNNs demand post-deployment training, i.e., there is a need to train the deployed DNNs to achieve improved inference accuracy. It is usually not preferable to send the edge devices to the manufacturer for further training. There are two reasons: (i) Post-deployment training may take advantage of end users' private data, which creates privacy concerns if sending such data back to the device manufacturer; (ii) it is often physically difficult to retrieve the devices and send to the manufacturer. Unfortunately, existing IP protection schemes focus mainly on the inference phase and lack the ability to be extended to the training phase. For example, in our first scheme described above, we need to know the DRAM error distribution of each device to train a device-dependent DNN. However, the error distribution can only be implicitly applied to the weights by reducing the DRAM timing but is not explicitly visible to the end devices. This prevents the devices to generate the malfunctioning DNN after the weights got updated.

We have the new observation that the precision difference between Stochastic Computing (SC) format and binary format representations can also be used as fingerprints to protect DNN IPs [84]. Weights are stored in binary format outside the accelerator. When they are loaded into the accelerator, the weights will be converted into SC format with some precision errors. We use the same training method as in the previous scheme, but this time we use the precision errors instead. To make the precision error distribution also device dependent, we propose a novel binary to SC conversion method which transforms DRAM errors into the precision error.

This is the first scheme that supports IP protection for both training and inference on accelerators. At the same time, our optimized computing paradigm in SC domain also provides improvements in performance and energy consumption.

1.2.3 DNN Protection for Non-Volatile PIM Accelerators

As non-volatile PIM accelerators could provide higher computation parallelism and avoid massive data movement between the memory and computing unit, ReRAM is a good fit for DNN computing. However, how to protect the DNN IPs after their deployment on ReRAM accelerators remains a less addressed problem. Compared to CMOS accelerators, new challenges arise: i) ReRAM's non-volatility retains data after power is off, making the stored weights easier to be extracted out from the accelerator; ii) ReRAM's crossbar structure can only compute on cleartext data, making encryption on weights impossible.

We utilize the new 1T4R structure [79] together with SC to protect DNN weights on ReRAM crossbars [86]. The multiplication is performed in the SC format, and all weights and inputs are represented in the form of bit streams. With the 1T4R structure, the long bits stream of each weight can be truncated into multiple slices. During run time, the slices will be dynamically combined to reconstruct the bit stream for computing. Therefore, even if the adversary extracts all the content from the crossbars, the actual reconstructed bit streams are still hidden without knowing the recombine pattern. Only the recombine pattern needs to be stored as confidential information. Using these shuffled slices for protection also avoids the need for encryption, and still maintains the PIM capability of ReRAM.

Another contribution of this design is to reuse the same slices with different recombine patterns. This can be regarded as a model compression technique, which significantly reduces the model size after being deployed onto the ReRAM crossbars.

1.2.4 Model Compression for Non-Volatile PIM Accelerators

On the one hand, although ReRAM is denser than traditional SRAM and DRAM, it still faces the pressure of rapidly increasing DNN weight sizes. On the other hand, because of ReRAM's low endurance and slow write speed, existing ReRAM based DNN accelerators require weights to be statically mapped on the crossbars to avoid frequently programming the cells. All these issues require a large number of crossbars which increases the hardware cost of ReRAM based accelerator products, or demands model compression techniques to utilize the limited resources more efficiently.

Unfortunately, existing model compression methods (e.g., weight pruning [54] or weight quantization [35]) that exploit weight sparsity and weight redundancy can not be effectively applied to ReRAM based DNN accelerators. Pruning removes the weights whose values are close to zero. However, because ReRAM crossbar is a tightly coupled structure, exploiting the random and irregular distribution of zero weights requires complex control and peripheral circuits. Quantization uses a shorter bit width to represent weight values, e.g., using 8-bit fixed point instead of 32-bit floating point. However, bit width below eight usually incurs significant accuracy degradation [37, 40].

In addition to the model compression technique in our previous scheme discussed above, we propose a second model compression technique that could be applied on ReRAM based accelerators [85]. We flip the bits in crossbars so that multiple bit matrices can share the same crossbar. Because the bits in a crossbar are accessed uniformly, separately flipping each bit induces large metadata and complex control logic. Therefore, we propose to only flip the bits in the granularity of rows and/or columns in a crossbar. We first cluster similar bit matrices together and then flip them to match the cluster's centroid bit matrix. For each cluster, only the centroid bit matrix is stored in the accelerator while all other bit matrices will be reconstructed from the centroid bit matrix during inference. However, since the bits can only be flipped in the granularity of rows and columns, it is very likely that a bit matrix cannot be flipped to perfectly match the centroid bit matrix. We apply a post-flipping calibration method that only updates the distribution statistics of batch normalization layers to mitigate the precision loss.

1.3 Contributions

This thesis makes the following contributions:

- DNN related security issues have been widely studied in both research and industry. However, most of them focus on the robustness of the DNN's accuracy, while the security of DNN IP is less addressed. Instead, we provide a thorough investigation of this overlooked yet important issue and propose effective solutions.
- We propose a series of solutions that target various security requirements and different types of accelerators. Specifically, we propose the solutions based on the order of their threat model's restrictions, i.e., from the assumptions that the adversaries only have non-expert attack capabilities all the way to the scenarios with the strongest attacks. In this way, the users can choose different solutions according to their specific security requirements.
- Going further, we also extend the protection on accelerators that are based on emerging memory technologies. Because ReRAM's PIM computing can only operate on cleartext

data, existing encryption based protection methods are no longer applicable to protect weights stored in ReRAM crossbars. This thesis proposes the first DNN IP protection scheme that can be used in ReRAM PIM accelerators.

- Model compression is an important topic to make DNNs practical on resource constrained accelerators. However, ReRAM's crossbar is a tightly coupled structure, most state-of-the-art model compression methods can not be used in ReRAM accelerators. This thesis proposes two novel model compression schemes to fill this gap.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces the background knowledge used in the proposed designs. This chapter also presents related work on securing DNN IPs on accelerators. Chapter 3 describes the details of our first design (`AEP`), which protects DNN weights using hardware fingerprints. Chapter 4 presents our second design (`SCA`) – a more general solution that protects both DNN weights and intermediate results, and supports both training and inference. Chapter 5 describes our third design (`SRA`) that could protect the weights in SC format for ReRAM based accelerators. (`SRA`) also comes with a new model compression method that could effectively reduce the number of crossbars to store the deployed weights. To further reduce the resource demand on ReRAM based accelerators, Chapter 6 proposes another more general model compression method that does not require the weights to be in SC format.

2.0 BACKGROUND

In this chapter, we introduce some preliminary backgrounds that could help to understand our proposed protection schemes in later chapters. Section 2.1 introduces the basic concept of DNNs and the most common layers that are present in DNNs. Section 2.2 describes the DRAM/eDRAM structure which is the base memory technology in our `AEP` and `SCA` designs. Section 2.3 describes one type of the emerging non-volatile memories — ReRAM structure — which servers as the base memory technology of our `SRA` design. Section 2.4 gives the introduction of the Stochastic Computing technique which is utilized in our `SCA` and `SRA` designs to produce hardware fingerprints. Finally in the last part of this chapter, we also provide a brief literature review that are related to our work in Section 2.5.

2.1 Deep Neural Networks

A Deep Neural Network (DNN) is a layered structure that contains multiple layers of neurons. The outputs of one layer's neurons are used as the inputs to the following layer (except the last layer whose outputs are the DNN's outputs). The outputs of one layer are used as inputs to its next layer in the network structure. Therefore, these dynamically produced data between layers are also called intermediate results in this thesis.

A DNN layer can be one of various types. The most common used types are: convolutional layers, classifier layers, activation layers, pooling layers and batch normalization layers. If a DNN contains convolutional layers, it is also usually called Convolutional Neural Networks (CNNs). Since CNNs are the most popular DNNs and convolutional layers take the majority of computation time, we mainly target at CNNs in this thesis.

2.1.1 Convolutional Layers

The weights in a convolutional layer are composed of multiple 3-dimensional filters and a 1-dimensional vector. All the filters are organized as a 4-dimensional tensor ($K \times C \times R \times S$). K is the number of 3-dimensional filters. Each filter has C channels, each channel is a $R \times S$ matrix. Each filter has its corresponding bias (a scalar value). So, all the biases are organized as a 1-dimensional vector of length K.

The inputs to a convolutional layer are organized as a 4-dimensional tensor ($N \times C \times H \times W$). N is the number of inputs (also called batch size). C is the number of channels of one input, each channel is a $H \times W$ matrix.

The convolution is computed by sliding each filter over the H and W dimensions on the inputs. Each convolution step performs a multiply–accumulate operation (MAC) between the filter weights and the overlapped inputs to produce a single value, which will then add the bias of that filter to generate one output value, as shown in Equation 1. $\mathbf{O}$, $\mathbf{F}$, $\mathbf{I}$ and $\mathbf{b}$ are the output tensor, filter tensor, input tensor, and bias vector, respectively. All the outputs are also organized as a 4- dimensional tensor ($N \times K \times E \times F$). N is the number of outputs. Since the results produced by the same filter are put in the same channel in the outputs, the number of output channels is K, i.e., the same as the number of filters. E and F are the height and width of one output channel.

$$\begin{gathered}\mathbf{O}[n,k,e,f] = \sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1}\mathbf{F}[k,r,s]\mathbf{I}[h+r,w+s] + \mathbf{b}[k] \\ 0 \le n < N, 0 \le k < K, 0 \le c < C, 0 \le h < H, 0 \le w < W, 0 \le e < E, 0 \le f < F\end{gathered} \tag{1}$$

2.1.2 Classification Layers

A classification layer is named so because its purpose is to perform the classification task based on the features learned by previous layers. They are often located at the end of the layer structure. A classification layer also computes Equation 1, and can be viewed as a special case of convolutional layers. We can use the same notifications of convolutional layers. Each input is a 2-dimensional matrix, i.e., $\mathbf{I}^{N \times C \times 1 \times 1}$. Each output is also a 2-dimensional vector i.e., $\mathbf{O}^{N\times K\times 1\times 1}$. There is a full connection with the inputs and outputs, i.e., $\mathbf{F}^{K\times 1\times 1}$ and $\mathbf{b}^{K}$. Therefore, classification layers are also called fully connected layers.

2.1.3 Activation Layers

A DNN containing only convolutional layers and classification layers are still a linear model as all the computations are additions and multiplications. To solve more complex problems, the activation layers are used to introduce nonlinearity. An activation layer applies a nonlinear function, e.g., ReLU ($max(0, x)$), Sigmoid ($\frac{1}{1+e^{-x}}$), on each input value separately. So an activation layer is an element-wise operation and does not change the dimensions of the inputs.

2.1.4 Pooling Layers

Similar to convolutional layers, a pooling layer slides a filter along the width and height of its inputs. However, each filter is two dimensional (i.e., $C = 1$) so that it applies the filter on each channel of the inputs separately. Another difference is that there are no weights in a pooling layer's filter, it only computes the max or average over the covered input values.

2.1.5 Batch Normalization Layers

Batch normalization layers normalizes the inputs into a standard normal distribution by applying Equation 2.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2}$$

x is each single input value to this batch normalization layer. (μ) and (σ) are the mean and standard deviation computed from all the input values in the current channel. γ and β are two learnable parameters like the weights in convolutional and classifier layers. ϵ is a small constant defined prior the training. After applying Equation 2, the distribution of each output channel is normally distributed.

2.1.6 Training vs. Inference

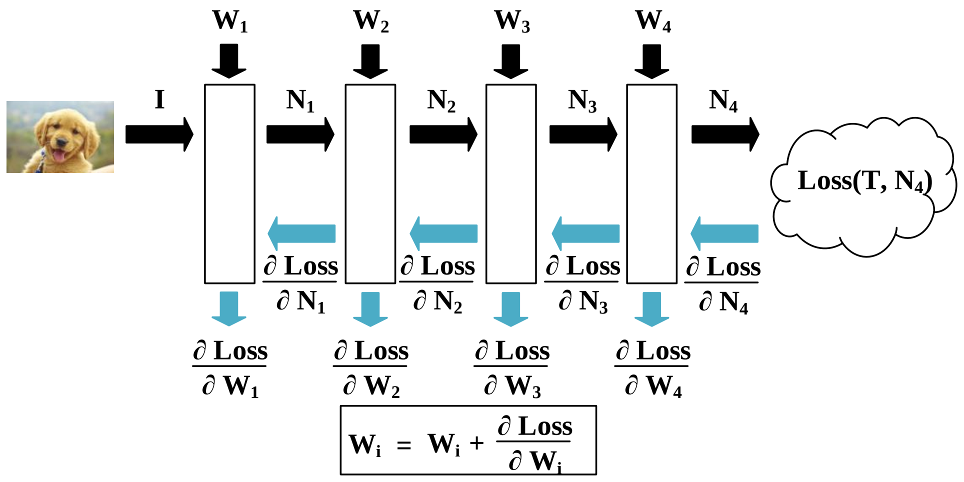

W_1
W_2
W_3
W_4
I
N_1
N_2
N_3
N_4
Loss(T, N_4)
$\frac{\partial \text{Loss}}{\partial N_1}$
$\frac{\partial \text{Loss}}{\partial N_2}$
$\frac{\partial \text{Loss}}{\partial N_3}$
$\frac{\partial \text{Loss}}{\partial N_4}$
$\frac{\partial \text{Loss}}{\partial W_1}$
$\frac{\partial \text{Loss}}{\partial W_2}$
$\frac{\partial \text{Loss}}{\partial W_3}$
$\frac{\partial \text{Loss}}{\partial W_4}$
$W_i = W_i + \frac{\partial \text{Loss}}{\partial W_i}$

Figure 1: The DNN training process.

A DNN has two different computing phases: *training* and *inference*. The training phase is to adjust the trainable weights of convolautional layers, classification layers and batch normalization layers to make the DNN fit a specific function. Then the trained weights are used in the inference phase to perform tasks, such as image classification, speech recognition, etc. Figure 1 shows the training process in an example 4-layer DNN that classifies images. The black arrows show the forward propagation pass. The first layer takes the image as inputs I together with this layer's weights W_1 to calculate the outputs N_1, which are then fed into the next layer. After the last layer computes its outputs N_4, the $Loss$ function calculates the difference between the network's outputs and the groundtruth T. Then the loss goes back through the network layer by layer in the backward propagation pass, denoted by the blue arrows. Take the last layer as an example, the inputs to this layer in the backward propagation pass are the gradients w.r.t. N_4. This layer computes the gradients w.r.t. its weights (i.e., $\frac{\partial Loss}{\partial W_4}$) as well as the inputs to this layer in the forward propagation pass (i.e., $\frac{\partial Loss}{\partial N_3}$). After all the layers finishes the backward propagation pass, each layer's weights will be updated by applying the equation $W_i = W_i - \eta \frac{\partial Loss}{\partial W_i}$ $(1 \leq i \leq 4)$, where η is the learning rate. The training phase is an iterative process, i.e., multiple rounds of forward propagation pass and backward propagation pass, until the loss stops decreasing.

Once the training phase is over, all the weights will be frozen. The inference phase uses the weights trained in the training phase, and only executes the forward propagation pass to classify an input image. The output of the last layer (N_4) is the predicted label of the input image.

2.1.7 Training with Approximation

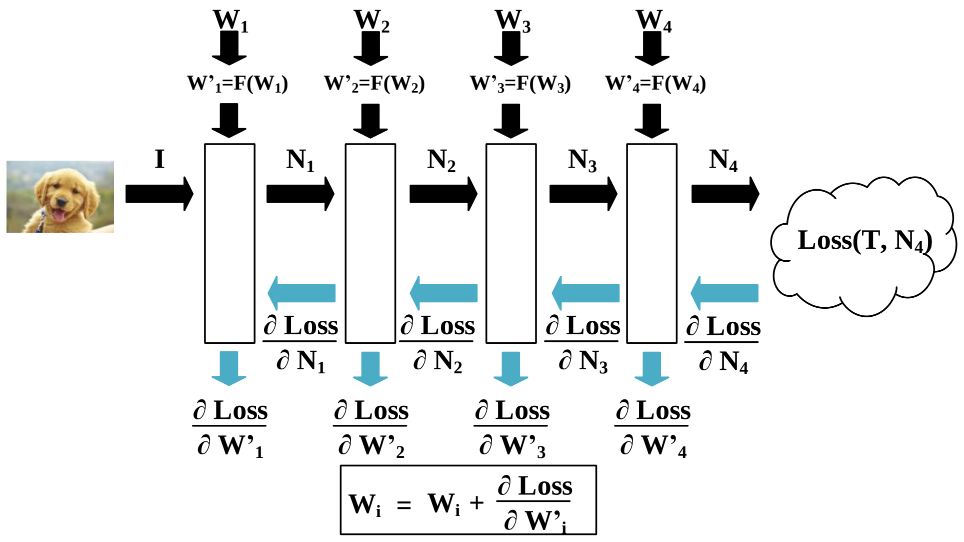

W_1
W_2
W_3
W_4
$W'_1=F(W_1)$
$W'_2=F(W_2)$
$W'_3=F(W_3)$
$W'_4=F(W_4)$
I
N_1
N_2
N_3
N_4
Loss(T, N_4)
$\frac{\partial Loss}{\partial N_1}$
$\frac{\partial Loss}{\partial N_2}$
$\frac{\partial Loss}{\partial N_3}$
$\frac{\partial Loss}{\partial N_4}$
$\frac{\partial Loss}{\partial W'_1}$
$\frac{\partial Loss}{\partial W'_2}$
$\frac{\partial Loss}{\partial W'_3}$
$\frac{\partial Loss}{\partial W'_4}$
$W_i = W_i + \frac{\partial Loss}{\partial W'_i}$

Figure 2: The DNN training process with approximation.

It has been proven that full precision computation (32/64-bit floating point) is not necessary in DNNs [29, 39]. Most hardware accelerators use fixed point representation for weights and/or inputs [14, 15]. Chen *et al.* [14] used different fixed point widths to train and inference DNNs. They found 16-bit fixed point is sufficient in inference phase, however, in order to make the training phase converge, at least 32-bit fixed point should be used in training. Courbariaux *et al.* [19] also investigated different low precision operations in training DNNs. They can achieve negligible accuracy loss with lower precision fixed point computations. Courbariaux *et al.* [20] then proposed to use binary weights (only 1 bit) in inference. The proposed training process in their work is shown in Figure 2. They keep two sets of weights, one is the full precision floating point version W_i, the other one is the low precision fixed point weights W_i'. They first binarize the weights through function F:

$$W_i' = F(W_i) = sign(W_i), (1 \leq i \leq 4)$$

and use W_i' in the forward propagation pass. During the backward propagation pass, they calculate the gradients w.r.t. W_i', but only use the gradients to update the full precision weights W_i. In inference phase, only W_i' are used, so the floating point weights W_i can be discarded. This binarized DNN incurs some accuracy drops compared with the full precision DNN, but significantly reduces the storage requirement, memory bandwidth demand, and the computation involved as multiplications can be completely eliminated.

2.2 DRAM/eDRAM

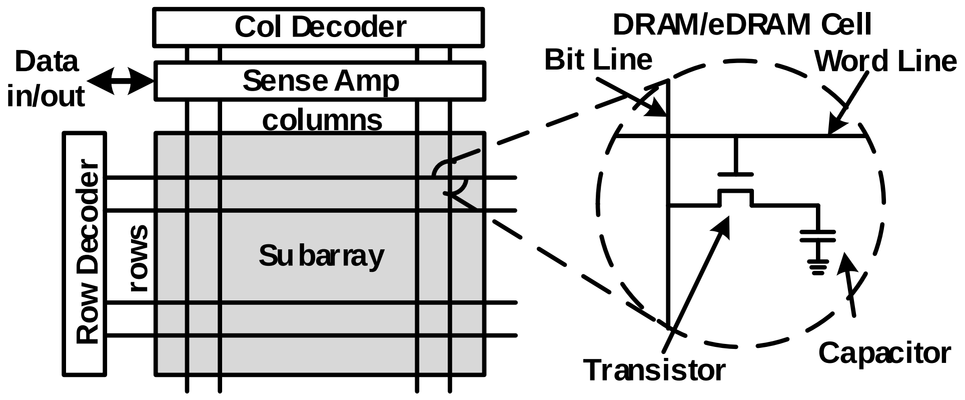


Figure 3: DRAM/eDRAM organization.

Dynamic random access memory (DRAM) is the most widely used type of memory in computer systems. Embedded DRAM (eDRAM) is the type of DRAM that is integrated on the same die of other computation logics (e.g., CPUs). DRAM and eDRAM have the similar structure, which is shown in Figure 3. The right part of Figure 3 shows the structure of one DRAM/eDRAM cell. A cell stores 1 bit data according to the amount of charge in the capacitor. The transistor controls the capacitor's connection with the bit line. During a read/write operations, the transistor connects the capacitor with the bit line, so that the charges stored in the capacitor can be shared with the bit line. Specifically, in a read operation, the bit line is first precharged to half of the predefined voltage, i.e., $\frac{V_t}{2}$. According to whether the cell contains charges, when the transistor connects the bit line and the cell, the bit line voltage will be raised up or dropped down because of charge sharing. Then the sense amplifier will detect the minor change on the bit line voltage to determine the value stored in the cell. As we can see, the read operation is destructive, i.e., the value stored in the cell is modified by the read operation. Therefore, a restore operation (similar to a write operation) is needed to write the original value back to the cell. In a write operation, the bit line is first programmed to V_t or 0. When the transistor connects the bit line and the cell, the value in the cell will be overwritten because the capacitance of the bit line is much larger than that of the cell.

The cells are organized as rectangular subarrays, consisting of rows and columns (left part of Figure 3). To access data in the subarray, the address first sends a subset of its bits (called the row address) to the row decoder, which will select an entire row from the subarray. The selected row will be read out and holded in the sense amplifier. Then another subset of the address bits (called the column address) is sent to the column decoder to select the target bits from the sense amplifier.

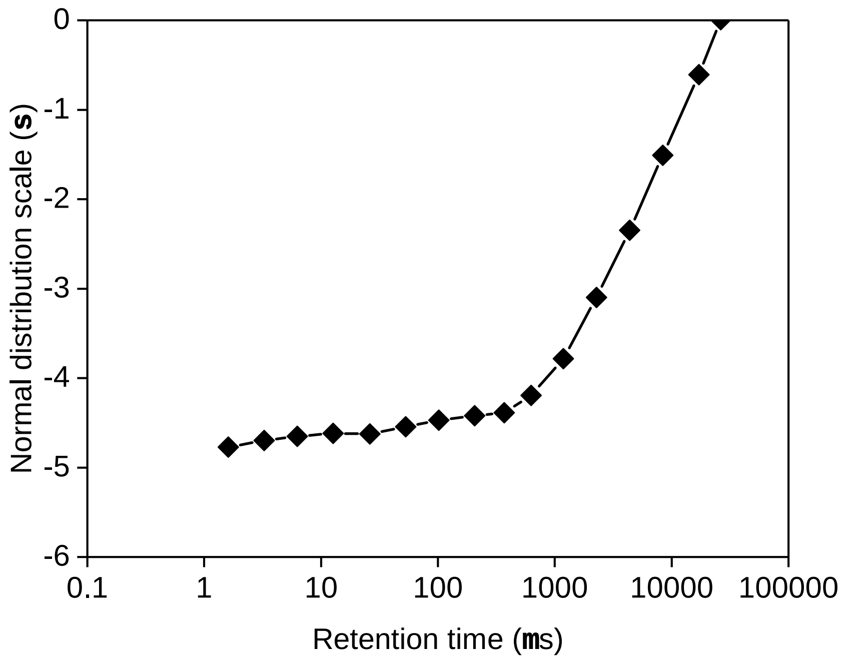


Figure 4: eDRAM cell retention time distribution [2].

Due to process variations, different DRAM/eDRAM cells leak charge at different speeds. The determining factors includes the size of the capacitor, gate length of the transistor, etc. It is impossible to make all these parameters exactly the for all the cells on a device. Usually, DRAM/eDRAM manufacturers follows the same timing standard to make their product compatible with each other. The standard (also known as JEDEC [1]) defines a set of conservative timings to guard the success of the operations. Violating the timing standard may induce data errors on the device. For example, recent studies showed that the bit error rates increase with reduced refresh frequency [2], as shown in Figure 4. Because weak cells leak charge much faster and the distribution of these weak cells are device dependent, many prior works [49, 17] have exploited it to device physical unclonable functions (PUFs) that map a set of input parameters to unique, device-specific signatures that can be generated repeatably and reliably.

2.3 ReRAM

2.3.1 ReRAM Basics and PIM Computing

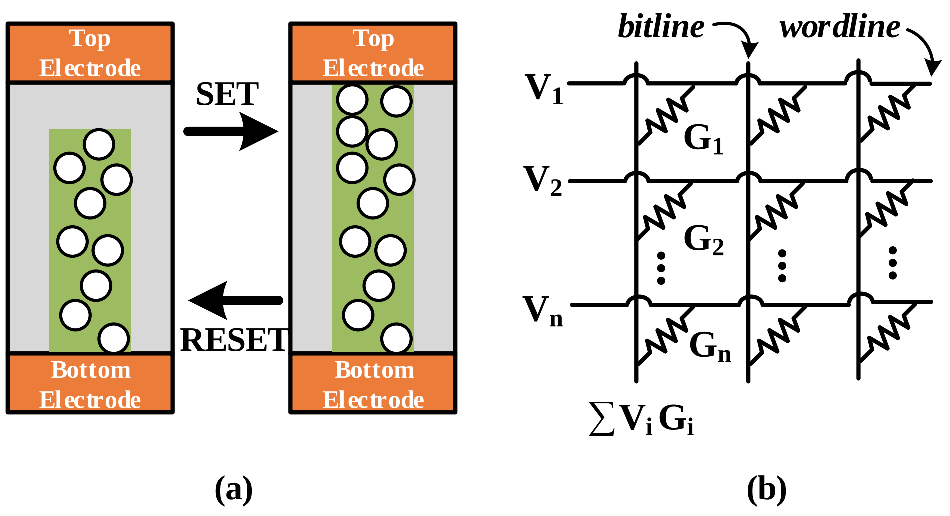


Figure 5: ReRAM and crossbar structure.

A ReRAM cell is a three-layer structure, consisting of a resistive switching layer sandwiched between the top and bottom electrodes, as shown in Figure 5(a). The information stored in a cell is determined by the resistance between the two electrodes. The cell's resistance can be programmed by *SET* and *RESET* operations to change the oxygen vacancy filament connecting the two electrodes. The oxygen vacancy filament determines the resistance between the two electrodes. Usually, a cell in the high resistance state stores 1, while a cell in low resistance state store 0. It is also possible to store more than 1 bit in a single ReRAM cell by dividing the resistance range into multiple levels to achieve higher density. However, this requires more complex I/O circuits and poses a challenge to programming accuracy. Therefore, many works [7, 85] use single-bit cells in PIM designs for practical concerns.

ReRAM cells can be organized in a compact crossbar structure to achieve the smallest $4F^2$ cell size. Figure 5(b) shows the matrix vector multiplication (MVM) operations taking place in a crossbar. The conductance (i.e., the reciprocal of the resistance) of the ReRAM cells are programmed according to the values in a matrix. The voltages of the word lines represents the values of a vector. Thus, the current flowing out from each bit line conveys the MVM result. This crossbar structure is an excellent match for DNN computing, whose core computation is MVM. The DNN weights are stored in the cells, while the input vector is converted into word line voltages. Compared to traditional Von Neumann architecture, the computation takes place inside the crossbar, avoiding the need to move the weight matrices for computation.

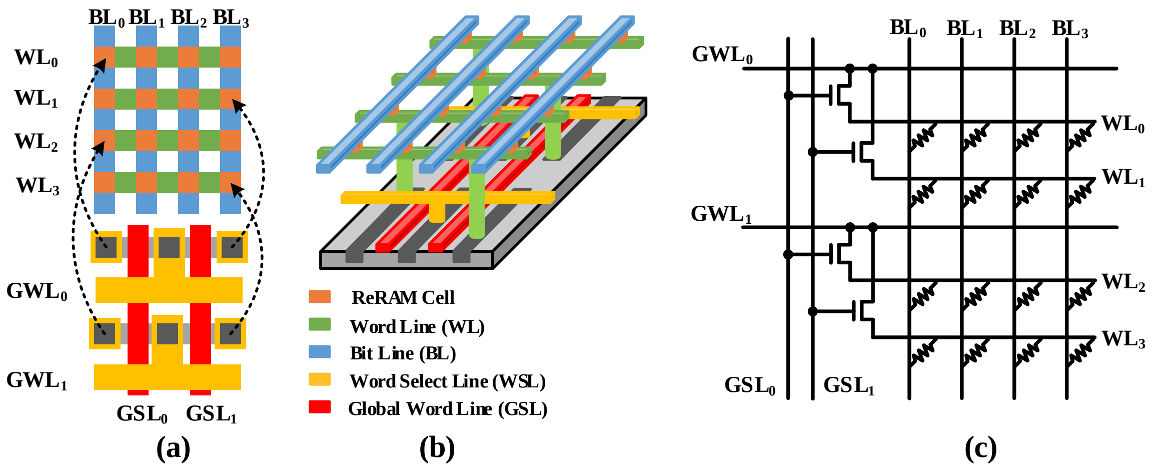
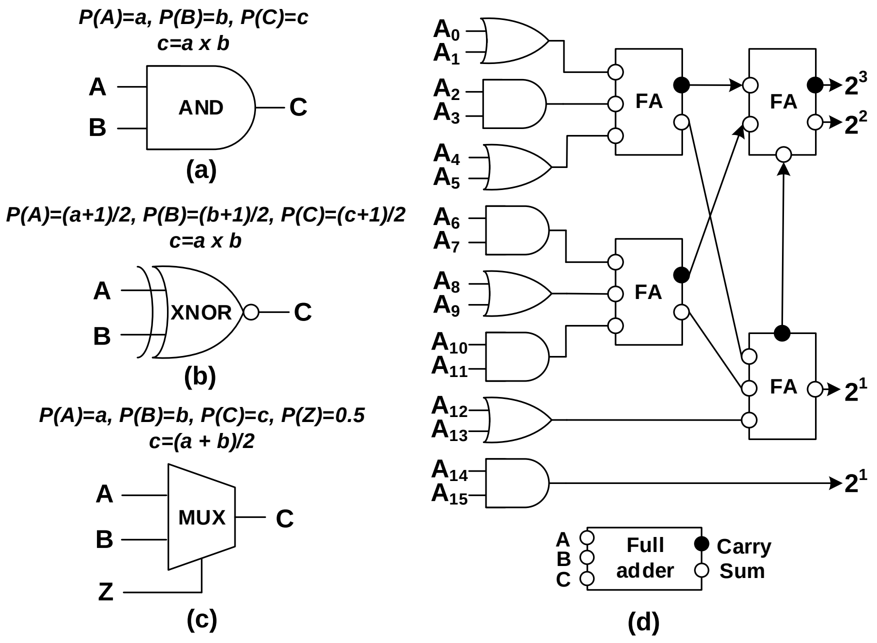


Figure 6: 1T4R structure.

Because there are no access transistors to isolate adjacent cells, unactivated word lines or bit lines can induce leakage current (called *sneak current*) during access. Recently, [79] presented a 1T4R ReRAM crossbar structure to address the sneak current problem. In a 1T4R crossbar, each 4×4 sub-array is isolated from other cells in the crossbar by 4 dedicated transistors, which are fabricated underneath the sub-array. Figure 6(a) and (b) illustrate the 2D and 3D layout, respectively. [79] fabricated a 1T4R test chip to show that the 4 transistors can be fully hidden underneath the area of the 4×4 ReRAM sub-array, thus still achieving the $4F^2$ cell size. Figure 6(c) shows the schematic diagram of one 4×4 sub-array. Besides bit lines (BLs) and word lines (WLs), there are also two global select lines (GSLs) and two global word lines (GWLs). The GSLs are used to select which rows in the sub-array to access. The GWLs connect to other sub-arrays horizontally in the crossbar.

2.4 Stochastic Computing



P(A)=a, P(B)=b, P(C)=c
c=a x b
A
B
AND
C
(a)
P(A)=(a+1)/2, P(B)=(b+1)/2, P(C)=(c+1)/2
c=a x b
A
B
XNOR
C
(b)
P(A)=a, P(B)=b, P(C)=c, P(Z)=0.5
c=(a + b)/2
A
B
MUX
C
Z
(c)
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15
FA
FA
FA
FA
2^3
2^2
2^1
2^1
A B C
Full adder
Carry
Sum
(d)

Figure 7: Stochastic Computing Implementation.

Stochastic Computing (SC) represents a number by using a random bit steam, in which the probability of the appearance of 1s indicates the represented value. For example, the bit stream $X = 011001001$ represents 0.4 because the probability $P(X = 1) = 0.4$. Since values are represented by probabilities, the representation for a value is not unique. A multiplication between two SC numbers can be implemented using bit-wise AND operations, as shown in Figure 7(a). However, only values in the range $[0, 1]$ can be encoded using the above method (called unipolar format). The bipolar format can represent values in the range $[-1, 1]$ by scaling it in $[0, 1]$. For example, the value $x = -0.4 \in [-1, 1]$ is first scaled to $y = (x+1)/2 = 0.3 \in [0, 1]$, then encoded into $P(Y = 1) = 00101010$. The multiplications for bipolar SC numbers are implemented using $XNOR$ gates (Figure 7(b)). For both unipolar and bipolar formats, additions are commonly calculated by multiplexers (MUXes), as shown in Figure 7(c). A third bit stream Z is required. $P(Z=1)$ is set to a constant 0.5 to give the same probability to select a bit from either A or B. So, the result is a scaled version of addition: $P(C) = P(Z)P(A) + (1 - P(Z))P(B) = \frac{1}{2}(P(A) + P(B))$. However, since half of the information in the input bit streams are lost, MUX based addition suffers accuracy reduction. Another way to perform additions more accurately is to use Approximate Parallel Counter (APC). APC counts the number of 1s in input bit streams with some errors to trade off logic gate counts. Figure 7(c) shows an example APC converting a 16-bit SC number to 4-bit binary number. Note that the output of APC is in binary format.

2.5 Related Work

With the prevalence of DNNs, many threats have emerged to DNN security. Accordingly, there are abundant defenses proposed in the literature to tackle those security issues. In this thesis, we classify these works into three categories based on which part of the DNN is the victim: algorithm validity, user privacy, and weight confidentiality. In this section, we present the most influential works that use architectural approaches for defenses.

2.5.1 Algorithm Validity

Attacks that target DNN's algorithm validity attempt to fool the DNNs with deceptive data. For example, adversaries may inject some noises into the input image to make it imperceptible to human beings, resulting in wrong outputs from the DNN. Szegedy *et al.* [70] is the first literature that found this phenomenon. Most of the attack and defense research fall in this category.

Wang *et al.* [74] proposes a heterogeneous architecture that couples a DNN accelerator with a CPU core to effectively detect adversary sample attacks. In addition to executing the original DNN task, the proposed architecture also simultaneously runs the detection algorithm. Guesmi *et al.* [26] proposes to use hardware-supported approximation as a defense strategy against adversarial attacks. They show that successful adversarial attacks have poor transferability to the approximate DNN implementation. The approximation is introduced by their approximate floating point multiplier, which also results in a significant reduction in power and delay. Fu *et al.* [24] proposes a precision-scalable accelerator that can randomly quantize the DNN into a different bit width to defend against adversarial attacks. The quantized DNN requires much cheaper computation in accelerators, thus more suitable in resource constraint settings such as Internet of Things (IoT) devices.

2.5.2 User Privacy

With advances in DNN architectures, it is becoming harder to train the DNNs. This is because of the massive number of weights, which make the DNNs much more expressive than the patterns present in the training data. It needs a great number of effort to prepare a sufficiently large dataset. For example, data are usually collected and preprocessed by a third party organization. Therefore, the collected data may contain commercial license restrictions or user privacy issues.

The most straightforward protection way is to use encryptions on the input data. However, instead of conventional encryption methods, Homomorphic Encryption (HE) is used to support computing on the encrypted data directly. However, HE introduces thousands of times computation overheads compared to raw data computing. Therefore, a lot of works have been proposed to design DNN accelerators that incorporate hardware HE supports [28, 23, 60]. To avoid the computation overhead of HE, Hashemi *et al.* [30] proposes to employ the trusted execution environments (TEE) for input privacy and computation integrity verification. Instead of purely relying on hardware to provide privacy guarantees, Mireshghallah *et al.* [53] propose to fine-tune the pretrained weights to make the DNN learn a special noise distribution which is injected in the input prior feeding to the DNN. The inputs are protected by the noise, while the DNN can still perform normal classification tasks on them.

2.5.3 Weight Confidentiality

The last category of attacks on DNNs is to stealing the DNN model itself, including the network architecture and the trained weights. Compared to network architecture, which is either widely known to the public or can be inferred by architectural hints [33], leaking the trained weights is more problematic as it directly relates to the previous two types of security issues. Firstly, knowing the DNN weights makes finding the adversarial samples easier by using gradient based methods [52, 11] (also known as white box attacks as opposed to black box attacks where the adversaries can only access the inputs and outputs of the DNNs). Secondly, it is possible to extract training data information from the trained weights [12], thus leaking the privacy of the training dataset.

Uchida *et al.* [71] embeds a watermark message inside the trained weights by adding some regularization terms during training. These watermarks can be later used to identify the ownership of the DNN. DeepMarks [13] takes a step further to identify the users who leak out the DNN model by embedding a unique binary code vector for each user as a fingerprint into the weights. These schemes require checking the weights to see whether the watermarks are present. However, retrieving the DNN from the user is sometimes infeasible. Guo *et al.* [27] proposes a similar idea, but instead of directly checking the weights, they use some special inputs to the DNN to trigger some predefined reaction which can be used to identify the authorship of the DNN. All these watermark based schemes can only identify the authorship of the DNNs, with no restriction on abusing them secretly, for instance, an adversary steals a DNN and only keeps it for personal use.

There are also encryption based works proposed in the literature to provide a higher secure guarantee, i.e., restricting the DNN only to work on target devices. P^{3}M [49] proposes a similar idea as our first design but behind our publication. P^{3}M extracts the memory startup error as a Physical Unclonable Function (PUF) to generate the key for encrypting the DNN weights. As PUF is unique for each device, the encrypted DNN can only be used in the target device. SFGE [10] proposes a sparse encryption scheme that only encrypts a subset of weights stored in ReRAM crossbars. The encrypted subset is determined by its impact on classification accuracy. To minimize the vulnerability window, only one layer will be decrypted to plaintext format at any given time.

The designs in this thesis also lie in this category and provide similar security guarantees to the above encryption based schemes. So, in the evaluation parts of our designs, we will further discuss the difference between our designs and those schemes. Integrating the protection of the previous two categories will be our future work.

3.0 PROTECT WEIGHTS USING HARDWARE FINGERPRINTS

In this Chapter, we propose our first design that protects the DNN weights on conventional CMOS technology based accelerators. We name this protection scheme `AEP` — an error-bearing neural network accelerator for energy efficiency and DNN protection. `AEP` provides an effective protection only on DNN weights, while our next design `SCA` (described in Chapter 4) extends the protection to both the weights and the intermediate results with a slightly more complex design compared to `AEP`. `AEP` is useful for small to medium sized DNNs, whose weights and intermediate results can be fully buffered in the main memory and on-chip buffers.

3.1 Threat Model

As the first step to design secure DNN accelerators, we do not assume the adversaries with extremely strong attack capabilities in `AEP`'s threat model. The extremely strong capabilities here refer to the attacks that require strong domain knowledge (e.g., access privileged data in main memory by compromising the OS or side channel attacks) or special hacking devices (e.g., SRAM probing attack by a laser probe). That being said, once the system is running, the main memory, the accelerator chip and the bus connecting these two components are in the secure domain, while anything stored in the persistent storage (such as the disk or flash memory) is vulnerable to attacks.

Generally, the GPU-pool based server needs to train a DNN (including its layer structure and the trained weights), which will then be deployed to a conventional CMOS based accelerator for inference at the client side. After the deployment, the client may pirate and/or tamper with the weights. A simple form of piracy would be directly copying the weights to another device, treating the DNN as a black box. Such threat is prevalent in our modern daily lives. An example is the DNNs deployed on the accelerators in autonomous driving cars. The car manufacturer owns the Intellectual Property (IP) of the DNN. The DNN is trained on the manufacturer's GPU-pool based servers or cloud GPU services purchased from some third party providers (such as Amazon Web Services (AWS), Google Cloud Platform, Alibaba Cloud, etc.). After the DNN is trained, it is deployed on the accelerators in their autonomous driving cars. Then the cars will be sold to the customers. If a competitor disguises as a customer and bought one of the cars, he/she may perform hacks into the storage of the car to extract the deployed DNN out and deploy it directly on their own products.

3.2 Protection Scheme

In this section, we first present the overview of `AEP` design and then elaborate the details of each component in `AEP`.

3.2.1 Overview

Figure 8 illustrates the overview of `AEP`. To deploy a DNN to a device (e.g., a car manufacturer may need to deploy an object tracking and recognition DNN to its autonomous driving car), the server (i.e., the car manufacturer in the example) extracts a device dependent memory error mask from each car that it manufactures, shown as ① in the figure. The server then integrates the error mask of the target device in the DNN training process (using GPUs), which generates a set of device dependent weights as part of the DNN, shown as ② in the figure. Next, the DNN and its weights are sent to the target device, which will employ the DNN to conduct inference tasks and achieve desired accuracy. However, while an attacker may port the weights to another device, shown as ③ in the figure, no matter if the pirate device has memory errors or not, the inference accuracy is below the acceptable threshold.

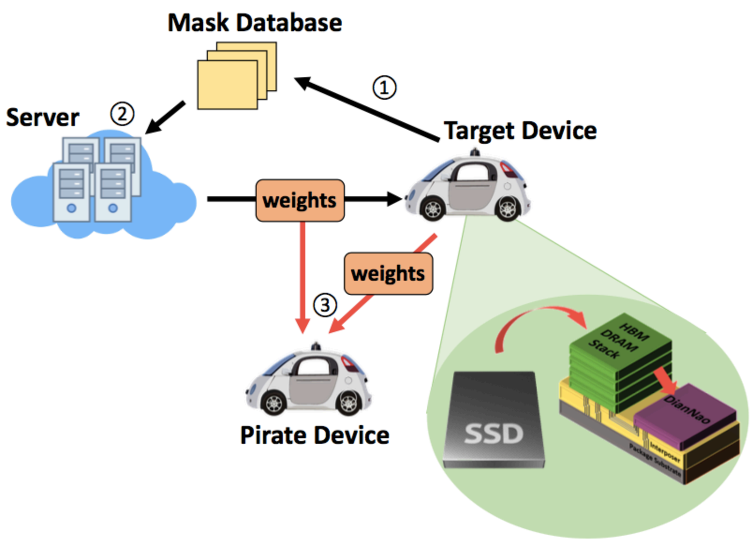


Figure 8: The overview of AEP design.

With the focus on IP protection in AEP, we assume the DNN is deployed before the target devices are released to clients. If there is a need to upgrade the DNN after release, the manufacturer needs to identify the corresponding error mask from its error mask database, and then generate new device dependent weights. There might be user privacy concerns in this process, which demands PKI based piracy enhancement [82]. We leave it to the future work.

We next elaborate AEP's design details.

3.2.2 Device Dependent Weight Matrices

Modern DNNs exhibit significant error resilience. As we described in Section 2.1.7, either replacing full precision weights with 16-bit fixed point values, or reducing refresh frequency to having a number of refresh errors [51], the DNN inference accuracy is often barely affected.

However, to make the AEP design possible, we need to solve a new problem that has not been studied before, i.e, ***is it possible to train device dependent DNNs that only produce satisfactory inference results on target devices but not on other devices***? To the best of our knowledge, we are the first to study this problem and our positive findings are highly valuable for real deployment.

Our baseline training process uses two sets of weights — the imprecise weights W_e used in forward computation, and the precise weights W_f used in weight update. W_e and W_f are similar to W_i' and W_i in Figure 2, respectively. Assuming we have a memory error mask M, a naive approach to train device dependent weights is to apply the mask when we are generating the imprecise weights. That is,

$$\begin{aligned} W_e &= f(W_f) = W_f \& M \\ W_f &= W_f + \frac{\partial Loss}{\partial W_f}. \end{aligned} \tag{3}$$

W_e is the set of weights that will be actually used in inference computing at the client side.

We tested the effectiveness of this strategy on benchmark `CNN1` and summarized the inference accuracy during training in Figure 9. The setting details can be found in Section 3.4. The `Normal` line denotes the training in the baseline. After five epochs, the training inference accuracy is close to 100%. The `withoutBN` line denotes the strategy that simply apply Equation (3) with a 5% error rate mask M to W_f during training. From the figure, we observe that this simple strategy does not work, i.e., `withoutBN` cannot improve the inference accuracy above 20%.

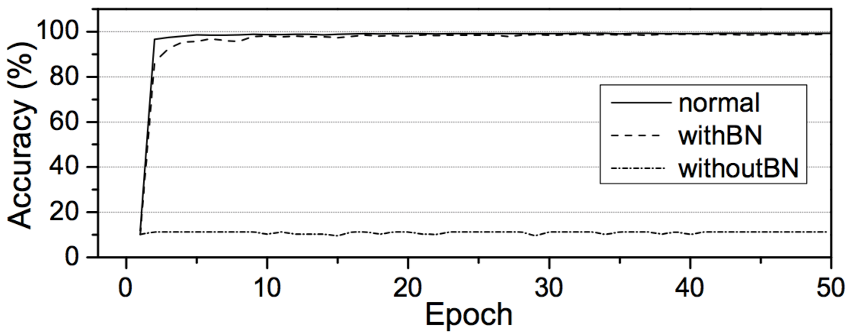


Figure 9: Training with and without batch normalization.

We then studied the training process and found that errors bits, when appearing in the integer bits of the weights, introduce large errors that cannot be corrected. For this reason, we adopt batch normalization [36] — a layer to mitigate the impact of bit errors in weights. That is

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta \tag{4}$$

where x is the output of the previous convolutional or classifier layer and input into the batch normalization layer. By subtracting the mean (μ) and dividing standard deviation (σ) from x, the input is standard normalized (with zero mean and unit deviation). γ and β are two learnable parameters like the weights in convolutional and classifier layers. ϵ is a small constant defined prior to the training.

We tested this new training process which adds batch normalization layers to the DNN. The `withBN` line in Figure 9 shows the effect of this strategy. As we can see from the figure, the loss of inference accuracy is negligible (0.22%) compared to `Normal`.

Given that the error mask is deeply integrated in the training process, we next check its dependency on the hardware devices.

After the training finishes, the server sends the weights W_f, instead of the weights W_e, to the target device. This is because the error mask M is extracted from the target device, as a result the target device can produce M and apply it on W_f using Equation 3 to generate W_e dynamically.

For IP protection, the weights W_f are visible to the attackers. Without knowing the secret mask M, the attackers have no access to the actual weights (i.e., W_e) that are used in inference computing. The attackers have two options to abuse W_f: one is to use W_f directly for inference on other devices; the other is to apply a different error mask M_X (the error mask of the pirate device), i.e.,

$$W_x \;=\; W_f \;\&\; M_X. \tag{5}$$

For the same benchmark `CNN1`, Table 1 summarizes the inference accuracy when adopting W_f on different devices. Since the target device can generate the error mask M, which can then be used to generate W_e, the accuracy is close the 100%, as shown in the second column of Table 1. No matter which abuse method described above is adopted by the attack, `CNN1` can only produce unacceptable inference accuracy, as shown in the last two columns of Table 1.

Table 1: Inference Accuracy on Different Devices

	Target device	Pirate device	Error-free device
`CNN1`	99.16%	10.15%	21.66%

3.2.3 Device Dependent Error Masks

The preceding section addresses the challenge of producing device dependent weights based on device dependent error masks. Next, we generate one such error mask as a proof of concept. In practice, there are many different approaches to generate device dependent error masks and signatures, e.g., those exploited in physical unclonable functions [68].

In `AEP`, our device dependent error masks are generated from DRAM restore errors. Given DRAM reads are destructive operations that destroy the stored values, they need to restore the values back to the cells after each read operation. JEDEC defines the timing that is required to reliably store the values in DRAM cells. There are two timing parameters (i.e., `tRAS` and `tWR`) that are directly related for restore/store the values for read/write operations, respectively. We only adjust `tRAS` to change the read restore time for a proof of concept for generating device dependent error masks.

The `tRAS` timing defined in JEDEC is often very conservative. Lee *et al.* [46] and Zhang *et al.* [81] proposed to safely reduce `tRAS` (and `tWR`) to improve DRAM performance, i.e., without introducing DRAM errors. In `AEP`, we propose to further reduce `tRAS` timing to introduce DRAM errors. Due to process variations, the amount of the DRAM errors and their distribution for a given `tRAS` are device dependent. Figure 10 presents the percentage of error bits for different `tRAS`. we adopted the same models as reported in [80, 81].

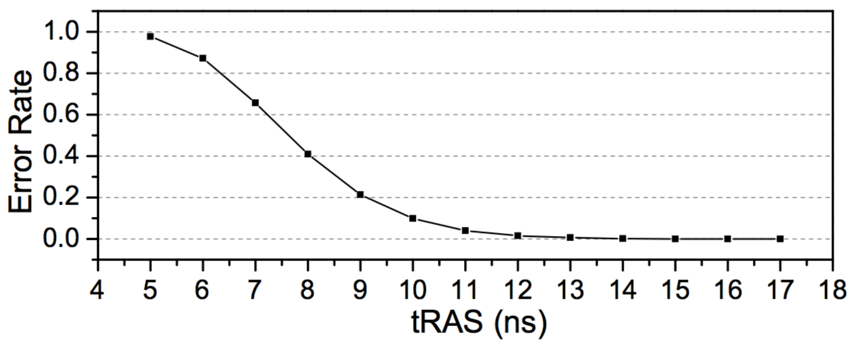


Figure 10: Reducing tRAS aggressively introduces memory errors.

Choosing tRAS timing is closely coupled with the error tolerance of the target DNN, i.e., the intrinsic layer structure and problem difficulty. As shown in Figure 11, different DNNs often exhibit different error tolerance. For high performance DNN applications, we often have a small bound on how much loss on inference accuracy can be tolerated. If we set the threshold to be 1%, CNN2 and CNN3 can tolerate 0.1% bit errors, MLPS, MLPM, and MLPL can tolerate up to 1% bit errors, while CNN1 can tolerate as high as 5% bit errors. For CNN1, we may reduce tRAS from 35ns in the baseline to 11ns, as shown in Figure 11.

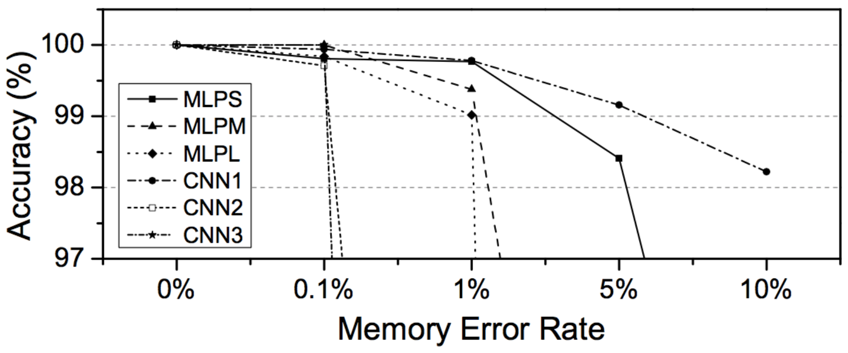


Figure 11: Different NNs have different error tolerance.

Recently, Song *et al.* [67] observed that different DNNs has different accuracy requirement, and for some applications higher accuracy is not always preferable. For example, the face detection DNNs in the camera apps on mobile devices favor more on timeliness over accuracy. Their observation is orthogonal to our design, i.e., while we set a fixed threshold (1%) in `AEP`, we will study DNNs with different thresholds in our future work.

While DRAM restore errors manifest as empty cells with no charge, a DRAM module may map cell state to 0 and 1 differently, i.e., some rows map no-charge cells as 1s while other rows map them as 0s. For this reason, the error mask consists of two submasks $M1$ and $M0$. They are used to identify the cells that are stuck at 1s and 0s, respectively.

The server took the post-fabrication test as shown in [46, 81] to extract the masks. This is done before deployment. Recent studies found that a small number of cells show variable retention time (VRT) [59], which may manifest as errors at different times. In our study, `AEP` proactively introduces 0.1% to 5% errors while VRT has a very low error rate, e.g., below 10^{-6} [59]. The impact from VRT is negligible.

3.2.4 Securing The Error Masks At The Client Side

In `AEP`, the DNN weights depend on the memory error masks and thus it is important to prevent attackers from extracting the masks from the client side. `AEP` defends the memory error masks using two approaches.

The first approach is that we disable the read path before release. The CMOS based accelerator and its DRAM module are packaged together such that only the accelerator can fetch data from the DRAM buffer to compute. A persistent attacker may probe the error mask by loading his/her own DNNs and correlating the outputs to the inputs to detect possible error bits. However, it is much more difficult and takes longer than memory march test [46].

The second approach is to adopt a light weight error mask selection mechanism as follows. We integrate a small table in the memory controller, e.g., the table in Figure 12 has 16-entries while each entry contains 3 bits. When accessing an DRAM row, we map its row address to an entry in the table, using the XOR of the address's last 4 bits with a 4-bit mask. The 3-bit entry content, e.g., '010_b'=2 indicates how to adjust `tRAS` at runtime. For example, if `tRAS` is set to 13ns globally, the memory controller effectively reduces `tRAS` for this row to

11ns if the mapped table entry contains 2; and there is no adjustment if the entry contains 0. Note that because DNNs have regular memory access patterns which are predetermined, the table look up can be done before the access, thus it is not on the critical path.

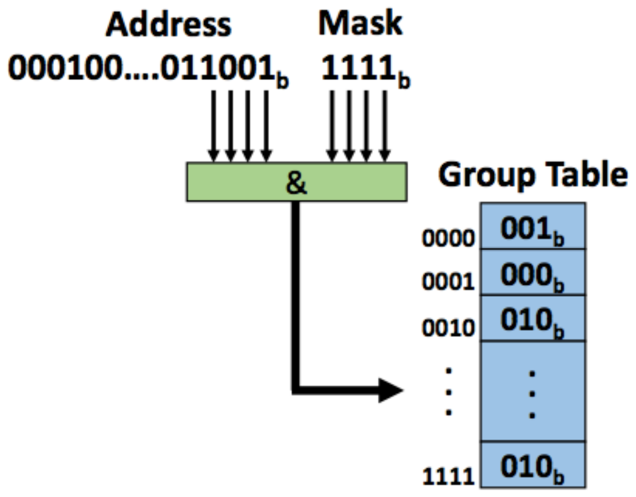


Figure 12: Dynamic `tRAS` adjustment.

The mask and the table only needs 7 bytes. It is sensitive information that is encrypted by the public key of the accelerator. The accelerator decrypts the mask/table before execution. In this way, even if the attacker can faithfully probe the error masks of the system, he/she cannot determine the actual error masks that are actually used in training.

The mask/table mechanism not only secures the error masks but also provides a better control of the error rate druing training. There are about 0.02% and 0.1% memory errors when we set `tRAS` to 16ns and 15ns, respectively. If a DNN has error tolerance at about 0.05%, choosing either `tRAS`=16 or `tRAS`=15 may not give us the optimal configuration. Instead, the server can set `tRAS` to be 16ns globally and then set the values of some entries in the above table to 0 while others to 1, which effectively set `tRAS` for these rows to 16ns while others being 15ns. The mix of these rows gives an average error rate that is 0.05%.

3.3 Accelerator Design

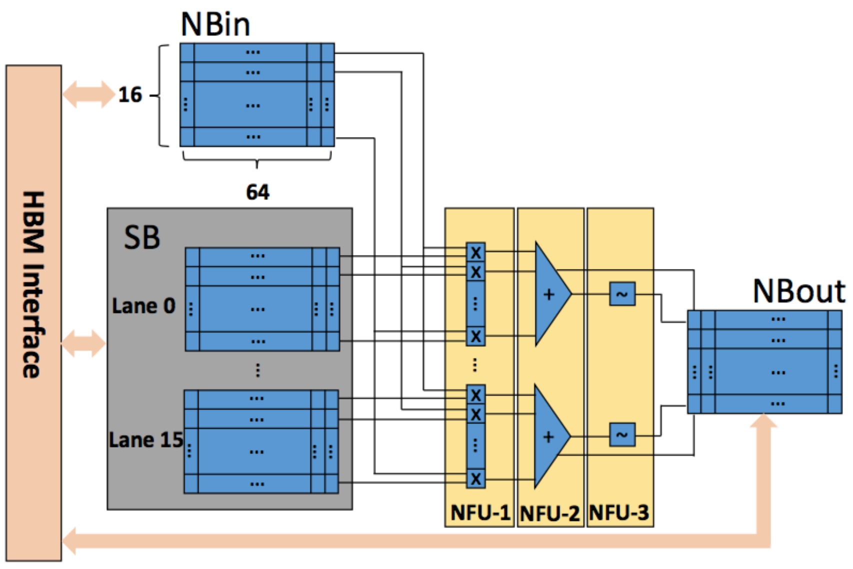


Figure 13: AEP accelerator.

AEP is an ASIC design that is built upon the DianNao [14] accelerator. Figure 13 shows the structure of AEP. It is composed of one Neural Functional Unit (NFU) and three SRAM buffers, i.e. input buffer (NBin), output buffer (NBout) and synaptic buffer (SB). The SB is divided into 16 lanes. There are 64 entries in NBin, NBout and each SB lane. One entry can store 16 16-bit fixed point data. First, the inputs of a layer are loaded into NBin, and the corresponding weights are loaded into SB. In each cycle, the 16 SB lanes provide 256 weights which are grouped into 16 16-weight groups to NFU. At the same time, the 16 input neurons from one NBin entry are broadcasted to each group. Every group of 16 neurons and 16 weights first multiply with each other in NFU-1 stage, then the 16 products are summed up in NFU-2 stage into one partial result and stored into NBout. If there is a partial sum of the same output neuron calculated in the previous cycle (buffered in NBout), it is also summed up in this stage. Once all the partial sums of an output neuron have been added together, the sum results goes through the NFU-3 stage to do a nonlinearity function, e.g. sigmoid. Thus, 16 output neuron's partial sums are computed in parallel. In order to support this high parallelism, Chen *et al.* [14] uses a memory system that can provide up to 250GB/s bandwidth, which is significantly higher than the capability of traditional DDR memories. In our experiments, we employ HMB [47] as the baseline memory system that provides 256GB/s bandwidth.

3.4 Experimental Methodology

To evaluate the effectiveness of our proposed scheme, we used the Theano [5] framework to explore the error tolerance of different DNNs. We tested three datasets with six DNN structures. `MNIST` [44] is a widely used gray scale image dataset for handwritten digit recognition. `SVHN` [56] is a real world house number recognition dataset obtained from Google Street View images. `Cifar10` [41] is a color image classification dataset containing 10 different object classes. We built 3 different DNNs on `MNIST`, `SVHN`, and `Cifar10`. In addition, we also built three different size multilayer perceptrons (`MLP-S/M/L`). Table 2 lists the detailed network structures.

Table 2: Datasets and Networks

Benchmark	Dataset	Neural Network
`MLPS`	`MNIST`	240-240-10
`MLPM`	`MNIST`	784-500-250-10
`MLPL`	`MNIST`	784-1500-1000-500-10
`CNN1`	`MNIST`	con5x20-pool2-conv5x50-pool2-500-10
`CNN2`	`SVHN`	conv5x32-pool3-conv4x64-pool3-1000-400-10
`CNN3`	`Cifar10`	conv4x32-pool3-conv4x32-pool3-conv3x64-conv3x64-conv3x64-pool3-500-250-10

For performance evaluation, we developed a cycle accurate simulator for the baseline `DianNao` and our proposed `AEP` design. We modified DRAMsim2 [62] to simulate HBM module according to [47]. The parameters of our simulator are listed in Table 3.

Table 3: Simulator Setup

DianNao	
NFU core	32nm 606MHz
NBin & NBout	2KB SRAM
SB	32KB SRAM
Data Format	16 bit fixed point (1 interger bit)
HBM	
Size	32GB 256GB/s
Timing	$\texttt{tRCD}$=12ns $\texttt{tWR}$=12ns $\texttt{tRP}$=12ns $\texttt{tRAS} = \begin{cases} 35\text{ns} & \texttt{DianNao} \\ 24\text{ns} & \texttt{AL} \\ 11\text{ns}/13\text{ns}/15\text{ns} & \texttt{AEP} \end{cases}$

In our evaluation, we compared three schemes.

- `DianNao` indicates the baseline accelerator proposed in [14]. It has no IP protection.
- `AL` indicates the scheme that reduces `tRAS` for performance improvement [46]. `AL` is built on top of `DianNao`.
- `AEP` indicates the design proposed in this paper. While both `AEP` and `AL` reduce `tRAS`, we compare them to study, in addition to DNN protection, what additional performance and energy benefits we may gain from aggressive reduction of `tRAS`.

3.5 Results And Analysis

3.5.1 IP Protection Effectiveness

We first evaluated the effectiveness of AEP in IP protection. We trained the DNNs using the device dependent error masks from one device and tested the trained weight matrices on error-free device as well as a device with different error masks. Table 4 summarizes our results from different benchmarks. The results are normalized to conventional traning and inference, i.e., the weights are trained for error-free devices and perform the inference also on error-free devices.

Table 4: Inference Accuracy with Different Error Masks

	Target Device	Pirate Device	Error-Free Device
MLPS	99.77%	39.35%	81.5%
MLPM	99.38%	28.31%	71.43%
MLPL	99.02%	10.41%	24.53%
CNN1	99.16%	10.15%	21.66%
CNN2	99.71%	89.94%	97.6%
CNN3	100%	60.95%	97.77%

From the table, we observed that the trained weights exhibit strong device dependency, i.e., while having negligible accuracy loss on the target device, they show unacceptable accuracy degradation on other devices (either with different error masks from the parite device, or directly used without performing any error masks).

3.5.2 Performance

In addition to IP protection, AEP improves inference performance due to the shortened memory access latency. Figure 14 compares the execution time of AL and AEP with the results normalized to DianNao. From the figure, AEP achieves, on average, 72% and 32% performance improvements over DianNao and AL, respectively. The large improvement comes mainly from tRAS reduction. Given all DNN benchmarks exhibit extremely high memory access intensity, reducing tRAS timing is very effective in improve inference performance.

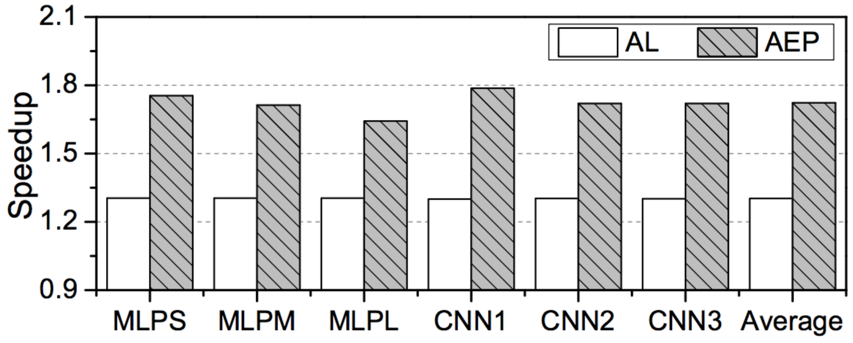


Figure 14: The performance comparison.

3.5.3 Energy Consumption

DRAM based memory is notorious for its refresh operations, which affect both performance and energy consumption. It is reported that refresh takes 20% energy of the entire memory system [6]. However, energy distribution is different in various DNN applications. DNNs are highly parallelizable, since the majority operations in DNNs are matrix multiplications. So the memory is more frequently read and written at all times. By reducing tRAS in read operations, both AL and AEP can improve energy consumption significantly. However, because AEP uitlizes the intrinsic high error tolerance in DNNs, tRAS can be further reduced to 11ns/13ns/15ns for different DNN structures (compared to the conservative 24ns in AL). As a result, AEP's energy consumption is only 56% of the baseline, while AL's energy consumption is 76% of the baseline.

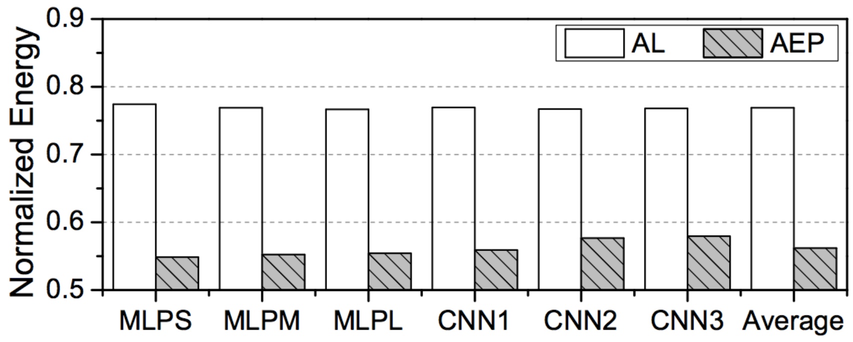


Figure 15: The energy consumption comparison.

3.5.4 Training Overhead

A big concern regarding training device dependent weight matrices is its training overhead. Figure 16 compares different training approaches. `Normal` indicates the traditional error-free training process. `Scratch` indicates the error-bearing training process, i.e., we integrate the device dependent error masks starting from scratch. `Continue` indicates the optimized training process that we integrate device dependent mask after `Normal`.

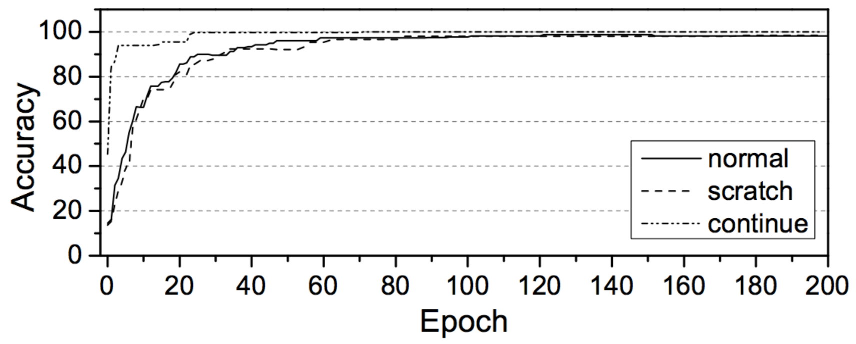


Figure 16: The server side training overhead.

From the figure, `Scratch` and `Normal` have comparable training speed (`Scratch` is slightly worse). However, `Continue` has much faster training speed than the other two. It converges in 24 epochs while the other two converge in 57 and 82 epochs, respectively.

The reason that we developed `Continue` is that, when deploying a DNN to multiple devices, we do not have to train the DNN from scratch for each device separately. Instead, we train the DNNs conventionally with no error masks until converge. Then, we use the pretrained weights as a starting point to integrate each device's error masks separately to get the each device's hardware dependent weight matrices. Adopting `Continue` can significantly reduce the training overhead at the server side.

3.5.5 AEP Robustness

The last experiment that we performed is to study the robustness of the design. In `AEP`, the error mask depends on the memory errors with different `tRAS` timing parameters. Due to process variations, the number of erroneous cells and their distribution are device dependent. However, due to VRT and chip voltage fluctuation at runtime, the real masks used in inference might show a small deviation from the profiled masks used for training.

Figure 17 studies the inference accuracy degradation due to the difference between the real masks and the profiled masks on the target device. The X axis indicates the percentage of difference; the Y axis shows the accuracy normalized to the DNNs that are trained on error-free weights.

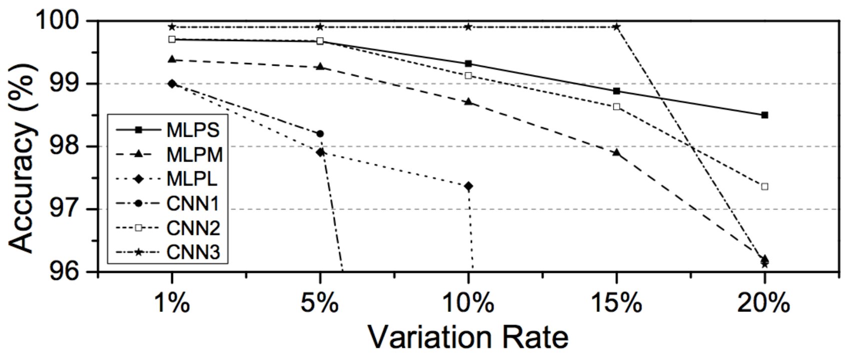


Figure 17: The `AEP` robustness.

From the figure, different DNNs have different tolerance levels on error mask deviation. For example, `CNN3` can tolerate up to 15% error mask deviation while `MLPL` shows a significant degradation with 5% difference. Recent study shows that the runtime deviation is often small, comparing to the number of proactively introduced errors by `tRAS` reduction. For example, VRT error rates are in the order of 10^{-6} [59], which have negligible impact on inference accuracy.

3.6 Conclusion

In this chapter, we introduced `AEP` — our first IP protection solution for DNN accelerators. `AEP` uses device dependent error masks to fine-tune the pretrained DNN weights. The resulting DNN can be guaranteed only functional on a specific authorized target device, while only producing inaccurate results on pirate devices. Our extensive experimental results show that `AEP` can tolerate 0.1% to 5% memory errors, which significantly reduces the timing restrictions on DRAM reads. As a result, `AEP` achieves an average of 72% performance improvement and 44% energy reduction over the `DianNao` baseline.

4.0 PROTECT WEIGHTS AND INPUTS USING HARDWARE FINGERPRINTS

Based on last chapter's discussion, `AEP` provides protection on DNN weights by automatically converting the low quality weights stored in persistent memory to high quality weights during loading them into the main memory. It can effectively protect small to medium sized DNNs when all the weights and the intermediate results can be hold in the main memory and on-chip buffers during the entire inference time. With the increasing size of modern DNNs and the emergence of Internet of Things (IoT) applications, new security requirements and risks arise in DNN accelerators for both training and inference.

Firstly, many DNNs demand post-deployment training, i.e., there is a need to train the deployed DNNs to achieve improved inference performance. It is usually not preferable to send the edge devices to the manufacturer for further training. There are two reasons. (i) Post-deployment training may take advantage of end users' private data, which creates privacy concerns if sending such data back to the device manufacturer. (ii) It is often physically difficult to retrieve the devices and send to the manufacturer. Unfortunately, `AEP` focuses only on the inference phase and is lack of the ability to extend to the training phase. Even if `AEP` expands its resources to support training, the newly updated weights will lost protection since all the data once loaded inside the accelerator are already applied with the error masks. Since the error masks are intrinsically stored in the memory, we can not explicitly apply the error masks reversely before writing the updated weights into persistent memory.

Secondly, besides the DNN weights, the intermediate results produced between layers also contain sensitive information of the DNN itself. For example, the intermediate results of fully connected layers and convolutonal layers are a linear function of the inputs, and knowing the inputs and outputs of a linear function makes inferring the weights a trivial problem. With the increasing DNN sizes in recent years, it is unavoidable to off-load the intermediate results to off-chip storage given limited on-chip resources on accelerators. However, `AEP`'s threat model is too strict such that the main memory is also included in secure domain. Although we could encrypt the intermediate results beforing off-loading them to the main memory and decrypt them before loading on-chip, due to the extremely high memory bandwidth demand in DNN accelerators, this approach introduces significant latency and energy consumption overheads.

In this chapter, we propose SCA, a secure DNN accelerator for both training and inference, to protect large sized DNNs whose intermediate results and newly updated weights need to be off-loaded to insecure off-chip memories.

4.1 Threat Model

SCA's threat model is more relaxed compared with that of AEP's. In SCA, only the accelerator chip is inside the secure domain, while everything outside the chip is vulnerable to attacks. Even if the attacker can compromise the OS or perform side channel attacks on the main memory, DNN weights are still kept confidential inside the accelerator.

The use case of SCA is similar to that of AEP's in terms of inference. The DNN is initially training on a server side GPU pool and then deployed onto the accelerators. The accelerators can be either purchased by customers or deployed in the public. The customers of the purchased device or anyone has physical access to the deployed devices in the public can be potential adversaries to hack into the device to extract the deployed DNN weights. However, different from AEP, SCA also supports training on accelerator while still provides protection on the updated weights when they need to be off-loaded to off-chip memories. This is useful in cases such as training on sensors deployed in the wild or training using user's privacy sensitive data on the client side, etc.

4.2 Protection Scheme

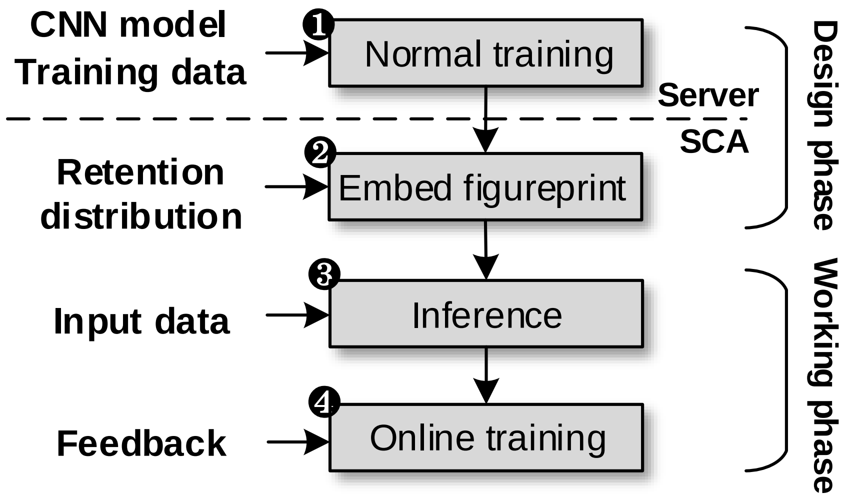


Figure 18: Overview of SCA's workflow.

A high level overview of SCA's workflow is shown in Figure 18. It consists of two phases — the *design phase* and the *working phase*.

In the *design phase*, machine learning experts train the DNN with their best efforts on the server side, i.e. using conventional full precision binary format weights and inputs. This step is denoted as ❶ in Figure 18. After this step, the trained weights are still in cleartext, which are vulnerable to attacks. In step ❷, we deploy the trained DNN to the target device. Given the target device, we first reduce the refresh frequency of the eDRAM buffers inside the accelerator to generate approximately 50% retention failures in some selected rows of the buffers. We record the this refresh frequency and the selected row addresses as the device's fingerprint, and store this information confidentially inside the accelerator. We then restore the default refresh frequency in all subsequent operations. The accelerator will run a few training iterations on the accelerator to embed the fingerprint into the weights, so that the fine-tuned weights are device dependent.

In the *working phase*, the user exploits the accelerator to accomplish its inference task (❸). In addition, the user may collect new inputs in the field and the desired inference results. Such data can be used to train the accelerator for improved inference accuracy (❹).

4.2.1 SC Based Protection

SCA computes in the SC domain, i.e, the weights and intermediate results participating in the DNN computing in the form of bit streams. In this section, we detail how to generate device dependent bit streams, so that the weights and intermediate results can integrate device's fingerprint after they are converted into this format. At the end of this section, we also propose an optimization technique to reduce the storage and computation overheads induced by SC's long bit streams.

4.2.1.1 Generating Device Dependent Bit Streams

Traditional SC generators (converting binary numbers into stochastic bit streams) use random number generators (RNGs) or linear feedback shift registers (LFSRs) to generate one bit per cycle. Given an N-bit binary number needs to be converted to a 2^N-bit stream to achieve sufficient precision, existing SC circuit designs tend to incur large area and performance overheads, e.g., the SC generator takes as much as 90% of the total area in [58].

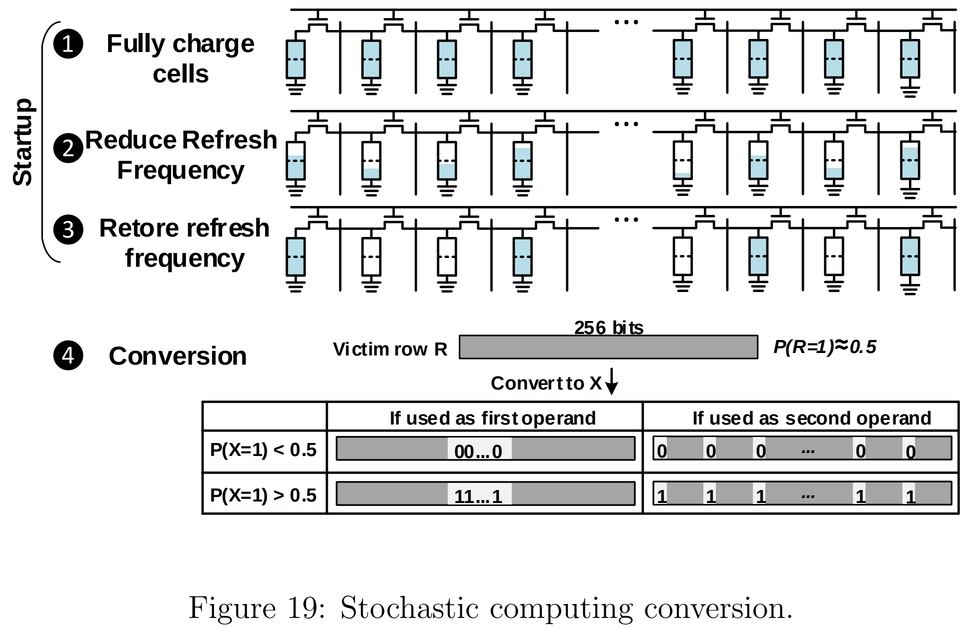


Figure 19: Stochastic computing conversion.

SC uses probability to represent values, i.e., it is the number of 1s rather than their positions that determine the overall computation accuracy. SC is intrinsically error tolerance as having one more or fewer bit incurs little impact. Consequently, instead of using RNGs or LFSRs, SCC leverages the PUF capability of eDRAM rows to generate device dependent random bit streams. As shown in Figure 19, when the system starts up, we first fully charge all the cells in some selected eDRAM rows (referred to as victim rows in the rest of this paper) (❶). Then we drastically reduce refresh frequency to make approximately half of the cells flip the stored value through capacitor leakage (❷). The reduced refresh frequency can be determined according to Figure 4. During normal DNN computing, `SCA` still uses the default refresh frequency (❸). The startup phase (❶-❸) only executes once. A victim row can be used multiple times to generate different SC numbers.

When converting a binary number to stochastic format, a portion of the bit stream is filled with 0s or 1s depending on the converted value. As shown in Figure 19 ❹, the victim row R is generated through steps ❶ to ❸, thus $P(R = 1) \approx 0.5$. If the probability of 1s in the target bit stream X is less than 0.5, 0s are filled into specific positions of the bit stream. Otherwise 1s are filled. The bits that are filled by 0s or 1s are similar to [48]. That is, if the generated bit stream is used as the first operand (e.g. weights), we fill consecutive 0s or 1s in the middle of the bit stream. If the generated bit stream is used as the second operand (e.g. inputs or intermediate results from the previous layer), 0s or 1s are scattered along the bit stream. The number of bits filled are determined by a look-up table.

4.2.1.2 Embedding Device Dependent Information in DNNs

DNNs are intrinsically error-tolerant algorithms because they are typically over parameterized. We modify the conventional training method to leverage this characteristic to embed the above introduced device dependent information into DNN weights. In each iteration of ❷ in Figure 18, the binary format weights W_b are first converted to stochastic format weights W_{sc}, which then participate in the forward phase to calculate the loss. In backward phase, gradients are calculated with respect to the binary format weights, i.e. $\frac{\partial Loss}{\partial W_b}$, as well as the update, i.e. $W_b = W_b - \eta\frac{\partial Loss}{\partial W_b}$. The basic principle is that DNN training only finds local minima on the loss surface, if a weight value is modified, other weights can change correspondingly to mitigate the impact and forces the training to find other local minima. This process is only performed for a few iterations (10 to 20) in `SCA`'s training.

4.2.1.3 DNN Protection

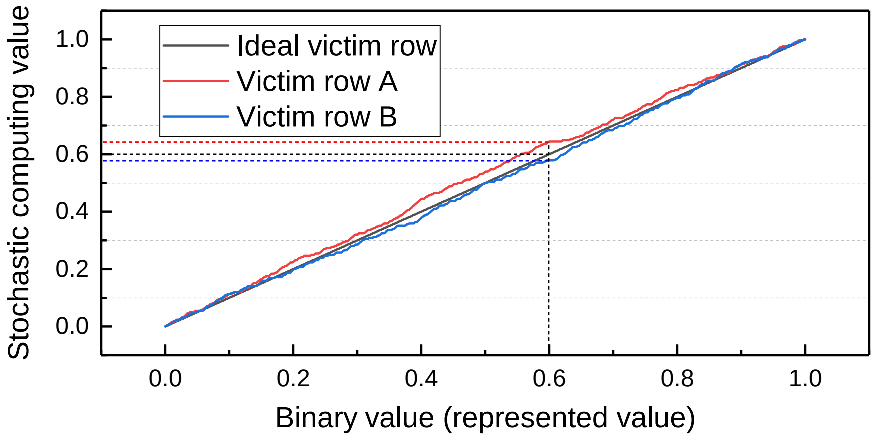


Figure 20: Stochastic computing values using different victim rows.

Because of process variation, relatively weak cells in the victim rows tend to leak charge faster during startup phase (as shown in Figure 19). As a result, the distribution of 1s along each row is device dependent. As a consequence, the converted stochastic value is also device dependent and usually not exactly the same as the binary value. Figure 20 shows the values converted from two different randomly selected victim rows. The X and Y axes are the binary value and converted stochastic value, respectively. We can see that different victim rows have different precision to represent the same value. Figure 20 also shows an example of converting 0.6 from binary format to stochastic format using these two victim rows. Assume the authorized device has victim row A. The actual weight value that is used in the DNN computing is 0.64. However, if an adversary extracts the value from either off-chip memory or on-chip eDRAM buffers, he/she can only get the binary value 0.6, which degrades the CNN prediction accuracy. Even if the adversary also use the same SC conversion method but on another device, because it is very unlikely that any of its victim row has the same error distribution as victim row A, the converted value may be 0.57 assuming victim row B is the selected row in the pirate device. Table 5 compares the inference accuracy of these three cases on the MNIST-MLPS benchmark, whose structure details are listed in Table 6.

Table 5: DNN Pretection

	On Same Device	**No SC Computation**	**On Another Device**
MNIST-MLPS	99.73%	57.98%	60.25%

Previous works ([83, 49]), once loaded on-chip, the weights are no longer under any protections since fingerprint information is removed by memory errors. So the weights can not be swapped out to off-chip memory if get updated in training. In SCA, whenever being swapped off-chip, the weights are converted back to binary format by APC, thus protected by the precision difference between stochastic and binary format value.

4.2.2 Robust Protection

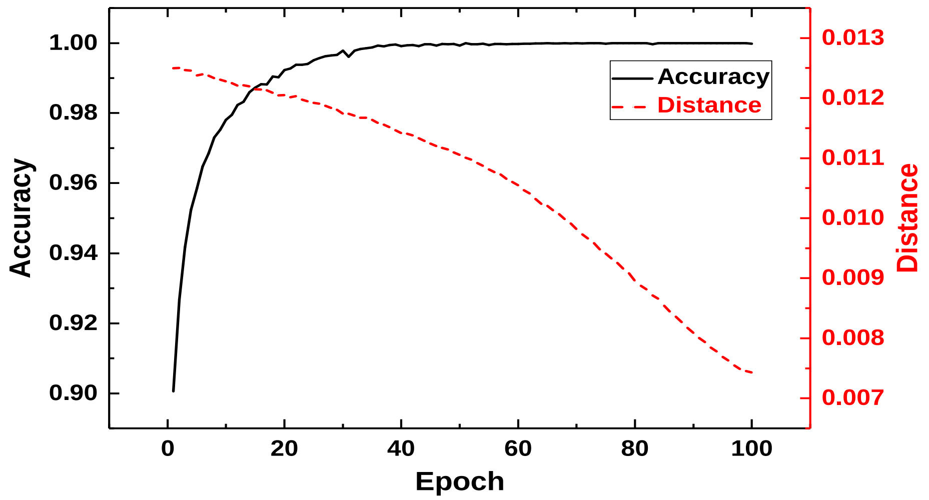


Figure 21: Accuracy and distance between weights of different formats for MNIST-MLPS.

Because DNNs are usually over parameterized, in addition to learning the classification task of the target problem, DNNs may also have the capability to learn the difference patterns between stochastic and binary format values. As a result, the binary format weight values (W_b) tends to be closer to the converted stochastic format weight values (W_{sc}) when sufficiently enough training iterations are applied. The lines in Figure 21 show the accuracy and 2-norm distance between values of these two formats with the training iteration proceeds on `MNIST-MLPS`. Eventually, the capability of protection scheme introduced in Section 4.2.1 vanishes, as shown by the red line in Figure 21.

To prevent the vanishing of differences between stochastic and binary values, we propose a remapping technique. For each weight conversion, instead of inserting into the middle of the bit stream, we use an offset to insert the consecutive 0s or 1s. The offset changes periodically overtime. However, a sudden change on all the weight conversion offset will drop DNN's accuracy. Therefore, we change a subset of weight conversion offset each time. More specifically, in each epoch we change one layer's weight conversion offset at a time. For example, in epoch one, we change the offset for the first layer's weight conversion while keep other layer's weight conversion offset unchanged. In epoch two, we only change the second layer's weight conversion offset, etc.

4.2.3 Space-Optimized Stochastic Computing

SC uses simple logic to perform computations, but it brings a new problem of exponentially increased data representation overhead which also aggravates the burden on memory bandwidth. For example, the weight matrix size of `AlexNet`, a state-of-the-art DNN, increases from 139MB in binary format to 238GB in stochastic format, which easily exceeds the capacity of on-chip memories in most DNN accelerators. In this section, we explore the optimizations on MAC operations to alleviate this problem.

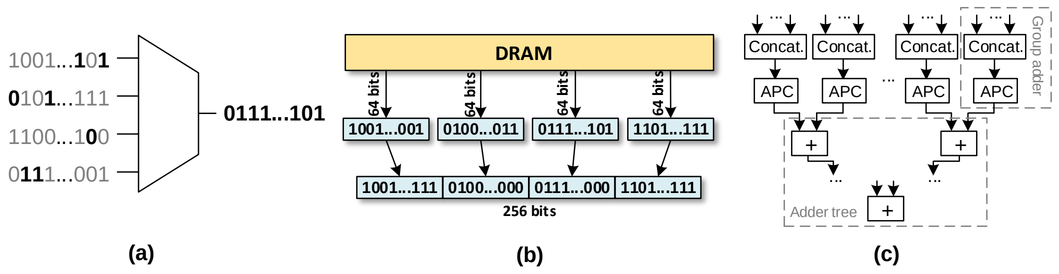


Figure 22: Convention and proposed SC Addition.

previous DNN accelerator designs using SC [48, 61] prefer APC based adders to MUX based adders due to their higher precision. The low precision of MUX based adders are caused by the dropped information (e.g., the grey bits in Figure 22(a) are dropped).

We propose a hybrid addition implementation that partition the summation inputs into groups. Inside each group, we first use MUX based adders to get the sum, which is then converted into binary format through an APC based adder. The binary format outputs of all groups then go through an adder tree to get the final sum. We tested different group sizes and found using group size of 4 can keep DNN's accuracy drop with 2%. Since the MUXes drop some bits, which is a waste of the previous computations (multiplications between weights and inputs in DNN). We can use shorter bit streams in multiplication, and use concatenation to substitute the MUX based adders. In the example shown in Figure 22(b), we use bit streams of 64 bits instead of 256 bits for group size of 4 to save memory bandwidth and computation. Figure 22(c) shows the entire MAC unit.

4.3 Accelerator Design

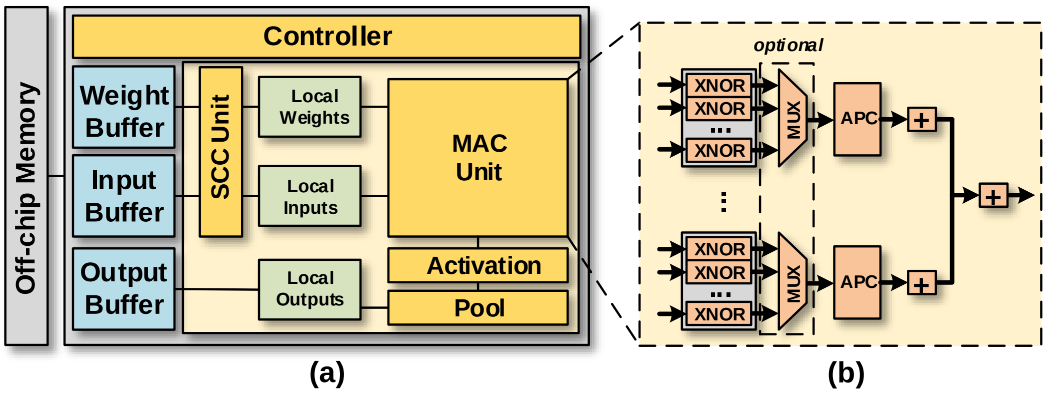


Figure 23: SCA accelerator.

A high level overview of the SCA architecture is shown in Figure 23(a). The fingerprint-embedded weights and the inputs/intermediate results are stored in binary format in off-chip memory before execution. For inference and training, SCA first loads the weights and the inputs/intermediate results into on-chip eDRAM buffers. To exploit stochastic computing (SC), the SC conversion (SCC) unit converts the buffered binary weights into stochastic format using the previously selected eDRAM rows, and store them in the Local Weights buffer. The MAC (multiply-accumulate) unit performs stochastic multiply-accumulation operations, shown in Figure 23(b). Finally, the results are written back to off-chip memory through Local Outputs and on-chip eDRAM output buffer. Whenever the weights and intermediate results between layers are in eDRAM buffers or off-chip memory, they are in binary format. In this way, the DNN is kept safe during the whole process.

Intuitively, since the fingerprint is embedded in the binary weights, conducting DNN inference using the binary weights on an error-free device leads to poor inference accuracy. The converted stochastic format weights carry device dependent fingerprint and thus produce accurate inference results only if the selected eDRAM rows match those on the target device.

4.4 Experimental Methodology

To evaluate SC's impact on DNN accuracy, we developed a custom DNN framework with C++ and CUDA and replaced all the computations in CUDA kernels to SC bit-wise operations. We tested on three widely adopted image classification datasets: `MNIST` [44], `SVHN` [56], and `Cifar10` [41]. In addition to DNNs for each dataset, we also include three multilayer perceptrons of different sizes (`MLP-S/M/L`) in our benchmarks. The details of the network structures are listed in Table 6.

Table 6: Datasets and Networks

Benchmark	Neural Network
`MNIST-MLPS`	250-10
`MNIST-MLPM`	500-250-10
`MNIST-MLPL`	1000-500-250-10
`MNIST-CNN`	con5x20-pool2-500-10
`SVHN-CNN`	conv5x32-pool2-conv5x64-pool2-256-10
`Cifar10-CNN`	conv4x32-pool2-conv5x32-pool2-conv5x64-pool2-500-10

We used Design Compiler with 45nm FreePDK library to synthesize the SC logic to get latency, area and power parameters. SRAM and eDRAM parameters were extracted from CACTI. We developed an in-house simulator to evaluate `SCA`'s performance and energy consumption.

We compare `SCA` with two baselines. `SC-CNN` [65] is a DNN accelerator implemented with stochastic computing with no security support. `AEP` [83] uses memory errors as hardware fingerprint to protect DNN weights, but no protection for training phase, and its computation is based on conventional binary arithmetic. We evaluate two schemes of `SCA`:

- `SCA-1` This is our basic `SCA` implementation as elaborated in Section 4.2.1.

- SCA-2 This implementation integrates the hybrid addition optimization.

We also investigate SCA's protection robustness in Section 4.5.3.

4.5 Results And Analysis

4.5.1 Characteristics

Table 7: SCA Characteristics

Units	**Number/Size**	**Area**	**Power**
Group Adder			
XNOR	256	$6553.6um^2$	3.53nW
APC	1	$3886.3um^2$	0.195uW
Total	1	$10439.9um^2$	0.195uW
MAC			
Group adder	2048	$21.38mm^2$	400.5uW
Adder tree	1	$1.71mm^2$	1.0328mW
Chip			
MAC	1	$23.1mm^2$	1.432mW
eDRAM Buffers	3×2MB	$17.6mm^2$	1.328W
SRAM Buffers	3×16KB	$0.204mm^2$	59.49mW
SC unit Total	1	$13.65mm^2$	233mW
Other	1	$3.97mm^2$	1.05W
Chip total	1	$59mm^2$	2.68W

Table 7 lists the characteristics of a SCA chip using 45nm technology. A SCA chip can accomadate 2048 group adders. Each group adder can multiply 4 stochastic numbers and sum up to one binary output. The outputs of all the 2048 group adders go through the adder tree to form the final summation. SCA uses three eDRAMs as on-chip buffers for input, weights and output, respectively. The converted stochastic numbers are stored in three SRAM buffers for faster access.

4.5.2 Protection On Weights

Table 8 shows SCA's protection effectiveness on inference. The DNNs are trained using the method described in Figure 18. The second column shows the inference accuracy on the same device as used for training (i.e. authorized devices). The third column shows the inference accuracy if the DNNs are running on pirate devices which cannot generate the same victim rows that are used in training. The last column shows the inference accuracy if the DNNs are used in conventional binary arithmetic based computing. All the accuracy numbers are normalized to that when training with binary format values.

Table 8: Inference Accuracy

	Target Device	Pirate Device	No SC
MNIST-MLPS	99.73%	57.98%	60.25%
MNIST-MLPM	99.69%	21.57%	24.75%
MNIST-MLPL	99.99%	20.99%	26.38%
MNIST-CNN	99.07%	14.27%	81.82%
SVHN-CNN	99.84%	69.04%	60.9%
Cifar10-CNN	99.99%	41.86%	52.44%

From the last two columns of Table 8, we can see a significant drop on all the benchmarks. No matter the pirate devices use the same SC conversion method or use binary arithmetic directly, the tested benchmarks can not generate satisfiable inference accuracy, which renders the effectiveness of SCA's weight protection scheme. When the tested benchmarks run on the same devices as used in training, only a negligible inference accuracy loss (less than 1%) is observed (first column in Table 8). This proves the fidelity of `SCA`'s protection scheme.

4.5.3 Protection robustness in Training

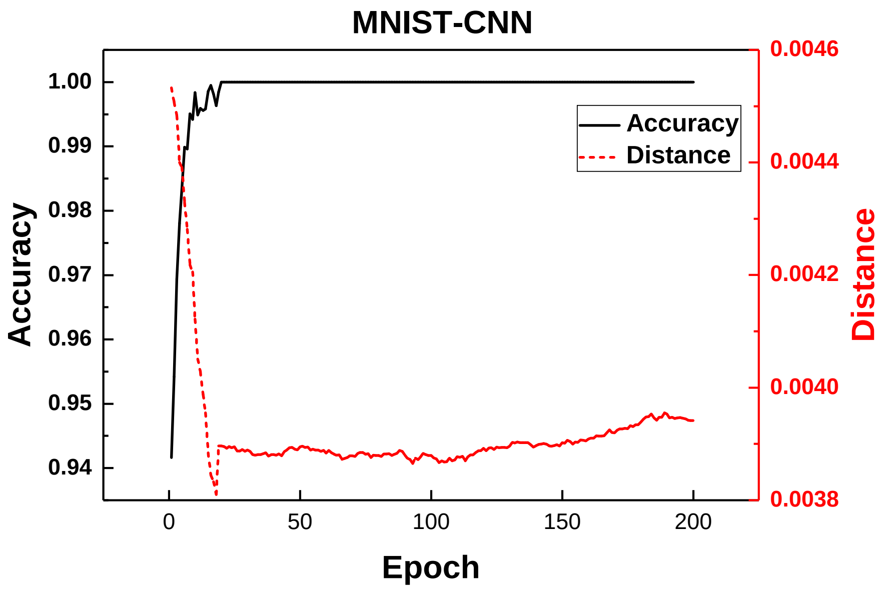


Figure 24: Protection robustness in training.

Figure 24 shows the effectiveness of our remapping technique for training. The solid black line shows the test accuracy normalized to conventional binary arithmetic based DNN. The dashed red line shows the distance between stochastic format weights and binary format weights. The first few epochs, where accuracy experiences an observable increase, show when the DNNs are learning the classification tasks. At first the distance is big due to the random initialization of weights. The distance drops with the training going on due to the excessive learning capability of the DNN to also learn the value differences between stochastic and binary formats. When the training reach the highest accuracy (convergence), the distance stops decreasing thanks to the periodically changed remapping technique for SC conversion. Because all the weights will use a new conversion offset after some epochs, the training can not find a constant difference pattern between the stochastic and binary format values.

Other benchmarks show similar trends as MNIST-CNN.

4.5.4 Performance

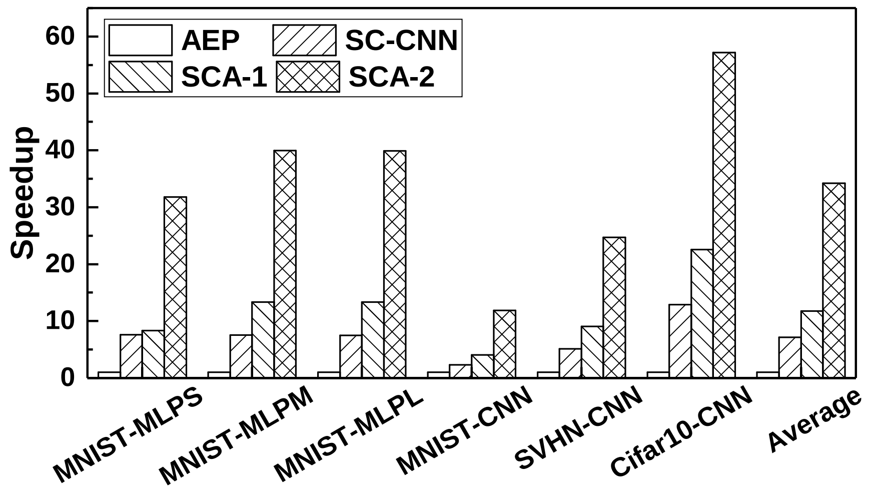


Figure 25: Speedup.

Figure 25 shows the speedups normalized to the AEP baseline. AEP uses conventional binary arithmetic for computation. The speedup of SC-CNN mainly comes from the faster SC based multiplications. Although SC-CNN employs a counter to replace the RNG for one of the multiplication operands, the other operand still needs a RNG to make sure the bit stream is randomized. On contrary, SCA-1 generates SC numbers for both multiplication operands by simple memory read and table lookup. So a larger speedup can be observed for benchmarks that require more data conversion. For example, Cifar10-CNN and MNIST-MLPL have more weights than other benchmarks, so they have higher speedups than other benchmarks. On averagae, SCA-1 achieves 11.7× speedup over AEP. SCA-2 uses shorter bit streams (64 bits instead of 256 bits), so both computation and memory access can be saved. Thus, SCA-2 can achieve 34.2× speedup on average. Compared to SC-CNN, SCA-2 achieves 4.8X speedup.

4.5.5 Energy

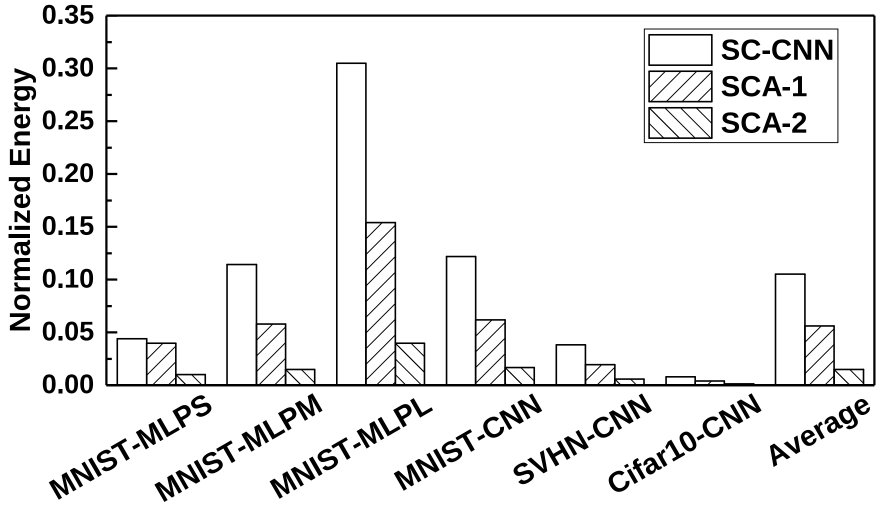


Figure 26: Normalized energy consumption.

Figure 26 shows the results normalized to AEP. SC-CNN saves energy by replacing multipliers with simple gates. On average, SC-CNN's total energy consumption is 10.5% of AEP. SCA-1 integrates SC conversion into the memory read process. Because SC conversion is needed for both input and weights whenever they are read from off-chip memory, SCA-1 further reduces the energy consumption to 5.6% of AEP. SCA-2 not only eliminates MUXes in addition operations, but also reduces the memory bandwidth burden by using shorter bit streams, thus SCA-2 only consumes 1.4% energy of AEP and 15.7% energy of SC-CNN.

4.5.6 Comparing with P^{3}M

P^{3}M [49] is a similar work that also uses on-chip eDRAM startup errors to protect DNN weights in accelerators. The protection method in P^{3}M is very similar to our AEP design, i.e., protecting the weights by memory errors. As described at the beginning of this chapter, directly moving the error mask applying step inside the chip cannot solve the security issues on intermediate results and updated weights. This limits P^{3}M only in executing small sized DNNs, in which intermediate results have to be fully buffered on-chip during the execution.

In addition, P^{3}M can not support training because of the lack of protection on updated weights. On the contrary, SCA does not have these limitations.

Another benefit of SCA is using SC for computing, which only uses simple logic to implement multiplication and addition. With our optimized MAC unit, the storage overhead can also be effectively addressed, at the same time further improving performance and energy consumption.

4.6 Conclusion

In this chapter, we introduced SCA our second IP protection solution for DNN accelerators. In SCA, DNN weights are in binary format when stored in off-chip memories, while in SC format when stored in on-chip memory. SCA uses the precision differences between these two formats to hide the actual weights values from the attackers.

SCA is an improvement over the previous AEP scheme in the following aspects: i) in addition to protection DNN weights, SCA could also protect intermediate results; ii) SCA reduces the secure domain to only contain the accelerator chip; iii) SCA provides protection for both training and inference on the accelerator. Our experimental results show that SCA effectively prevents pirating the DNN IP from the authorized devices. In addition, it achieves 4.8× and 34.2× speedups and 84.3% and 98.5% energy reductions over a non-secure baseline and an inference-only secure baseline, respectively.

5.0 PROTECT WEIGHTS AND INPUTS IN RERAM CROSSBARS

Besides the designs based on conventional CMOS technology [15, 16, 38], existing works have also demonstrated the use of emerging non-volatile technologies, such as metal-oxide resistive random access memory (ReRAM) [75], spin-transfer torque magnetic RAM (STT-RAM) [72], and phase change memory (PCM) [9], to design process-in-memory (PIM) computing engines for DNNs. Among them, the ReRAM crossbar structure is the most widely studied approach. By performing the computation in memory, the massive data movement can be avoided. Also, the crossbar based PIM paradigm provides opportunities for much higher computation parallelism.

However, how to protect the DNNs after their deployment on ReRAM based accelerators remains a barely addressed problem. Firstly, due to its non-volatility, ReRAM does not need a continuous power supply to retain data. This makes the accelerator susceptible to new security vulnerabilities, for example, accessibility to the stored DNN if a device gets stolen. Secondly, because ReRAM's crossbar structure can only compute on plaintext data, it is not feasible to store the encrypted weights on the crossbars. Recently, there are previous works [83, 84, 49] that utilize hardware characteristics as fingerprints to protect DNNs. Unfortunately, they can only protect the DNNs stored off-chip, thus only applicable to CMOS based accelerators. Once the DNN is loaded on-chip, the weights are automatically converted into plaintext format, making the entire DNN at risk if the accelerator is based on ReRAM technology.

In this chapter, we propose `SCA` — a secure ReRAM based DNN accelerator that stores DNN weights in an encrypted format while maintaining ReRAM's PIM capability. The multiplication is performed in the Stochastic Computing (SC) format, and all weights and inputs are represented in the form of bit streams. `SCA` uses the 1T4T crossbar structure [79] so that each physical bit line is sliced into segments, which can be activated separately. An arbitrary set of segments from a pair of two adjacent physical bit lines can be merged into a logical bit line, which stores the bit stream of a weight. Therefore, the actual weight value is encrypted by keeping the merging pattern confidential. We also propose to encrypt multiple weights on the same pair of physical bit lines, reducing the storage overhead of long bit streams.

5.1 Threat Model

While `SCA`'s threat model is more relaxed than that of `AEP`'s, the threat model of `SRA`'s is the most relaxed among these three schemes. In `SRA`'s threat model, we do not assume the accelerator is inside the secure domain any more. Since `SRA` is a ReRAM based accelerator, it retains the weights even after the system is powered down. Thus, it is gives the attacker plenty of time (days to months) to steal the embedded DNN weights.

One such attack example is that the attacker can run the DNN on the accelerator once, and then unplug the accelerator from the chip and stream the data out from the ReRAM crossbars.

5.2 Protection Scheme

5.2.1 Overview

In `SRA`, both weights and inputs are computed in SC format. However, only positive numbers can be represented by the *unipolar format* introduced in Section 2.4. Although there is also a *bipolar format* of SC representation that could encode both negative and positive numbers, the multiplication needs to be performed by XNOR operations which are not applicable to crossbar implementation. Fortunately, the ReLU activation in the DNNs filters out all negative values in each layer's outputs (i.e., inputs to the next layer), only weights may have negative values. In `SRA`, we only store the absolute values of the weights in the *unipolar format* SC bit streams while the sign bits are stored is a separate crossbar (called the sign bit crossbar in this paper). It is possible to expose the sign bits to the adversary, but only knowing the sign bits without the actual absolute weight values can not generate useful inference results.

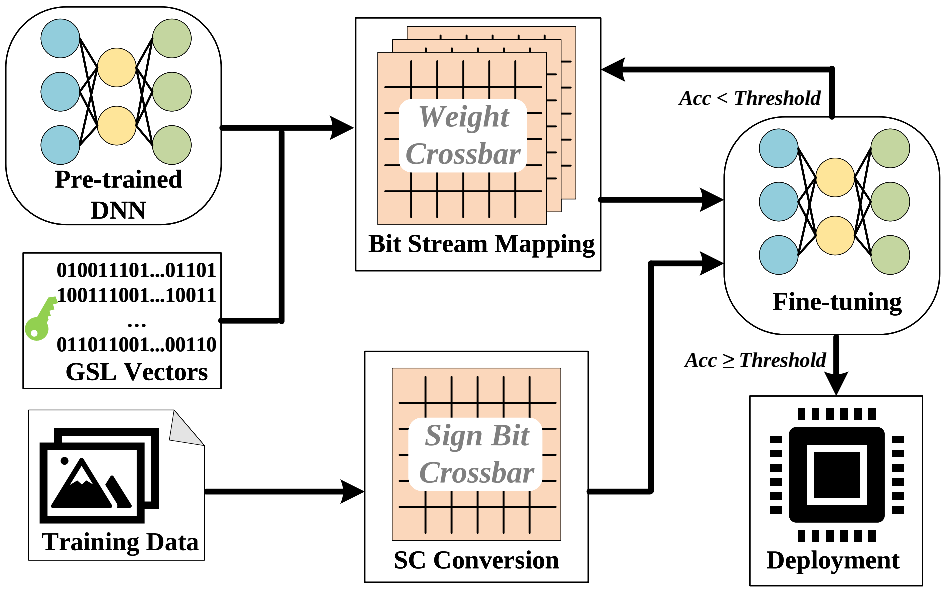


Figure 27: An overview of the `SRA` workflow.

Figure 27 shows the overview of `SRA`'s workflow, which takes a pre-trained DNN, the training data, and a set of *GSL* vectors as inputs. Each *GSL* vector determines how the segments are merged into logical bit lines. The *Bit Stream Mapping* step (Section 5.2.3) uses a mixed-integer linear programming (MIP) model to map the weights onto the logical bit lines. Because the mapping introduces noises into the weights, the DNN needs an iterative process of fine-tuning and mapping until the accuracy exceeds a predefined threshold. The *SC conversion* step (Section 5.2.4) converts each layer's inputs to SC format before computing with the weights. Note that all the *Bit Stream Mapping*, *SC conversion* and *Fine-tuning* steps are performed offline, and the ReRAM crossbars are only programmed once in the *Deployment* step.

5.2.2 Encrypted Crossbar Design

In order to slice the bit lines into segments, we adopt the rotated layout of the original 1T4R structure to switch the functionality of the bit lines and word lines. Since the crossbar is a symmetrical structure and the ReRAM cells are acting as resistors during computation, the rotated layout does not need any changes inside the crossbars, only the positions of the word line drivers and the ADCs are exchanged. As shown in Figure 28, the ADCs are placed at the right of the crossbar while the word line drivers are at the bottom. In this new layout, we rename the original horizontal global word lines (GWLs) to global bit lines (GBLs), and the original bit lines (BLs) are called word lines (WLs).

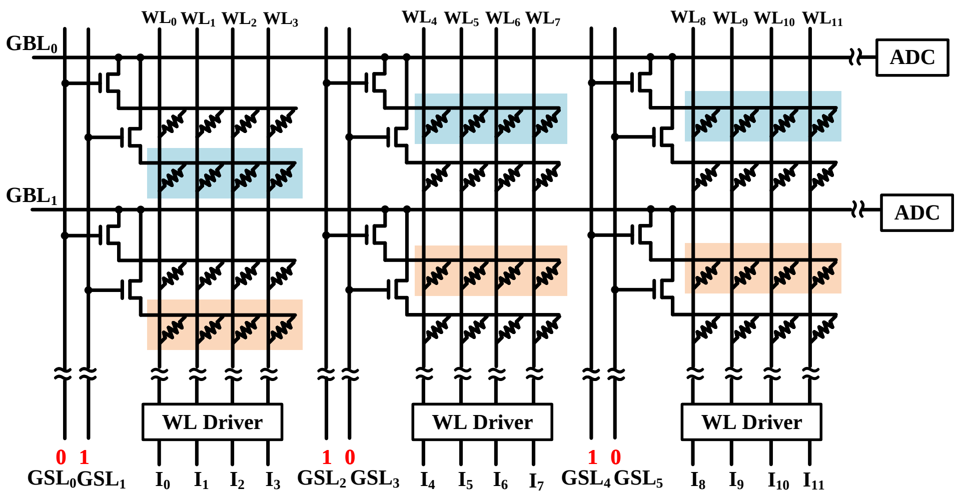


Figure 28: The rotated layout of the 1T4R structure.

In this new layout, each row in the crossbar is now divided into segments consisting of 4 cells. The GSLs control which segments are connected to the GBLs during computation. For the example shown in Figure 28, the first pair of GSLs selects the segments on the second and fourth rows to connect to the GBLs, the second pair of GSLs selects the segments on the first and third rows to connect to the GBLs, etc. As a result, all the blue cells in Figure 28 are merged to form a logical bit line to compute with the inputs I, and the results are accumulated on GBL_0. Similarly, all the orange cells in Figure 28 are merged to form another logical bit line to compute with the inputs I, and the results are accumulated on GBL_1. We restrict each pair of GSLs can only be 01 or 10, such that either the segments on the even rows or the segments on the odd rows will be connected to the GBLs. Therefore, only one bit in each GSL pair needs to be stored. In SRA, we use a bit vector to record the first bit in each GSL pair (i.e., GSL_0, GSL_2, ...). This bit vector is referred to as the GSL *vector* in SRA. The uncolored cells in the figure can form another two logical bit lines when all the GSLs negate their inputs. By keeping the GSL vector secret, the actual content of the logical bit lines is stored in the crossbar in an encrypted form.

5.2.3 Bit Stream Mapping

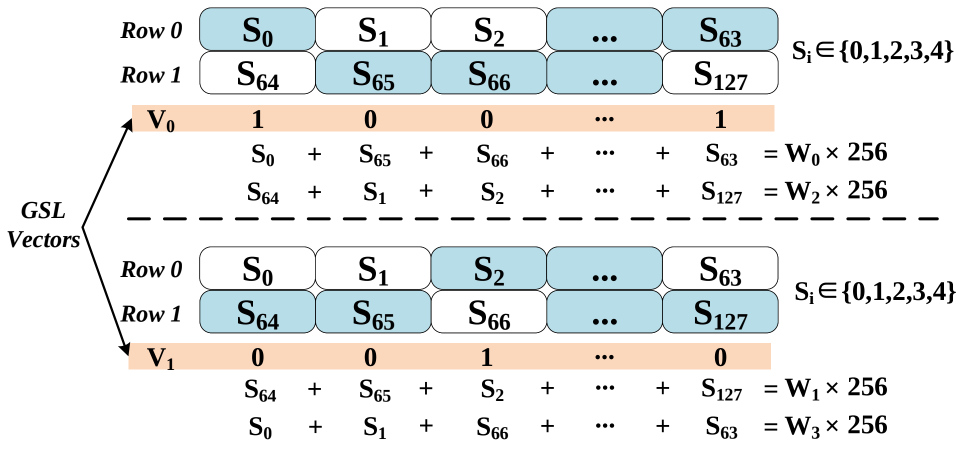

Row 0
Row 1
S_0 S_1 S_2 ... S_{63}
S_{64} S_{65} S_{66} ... S_{127}
$S_i \in \{0,1,2,3,4\}$
V_0 1 0 0 ··· 1
$S_0 + S_{65} + S_{66} + \cdots + S_{63} = W_0 \times 256$
$S_{64} + S_1 + S_2 + \cdots + S_{127} = W_2 \times 256$
GSL Vectors
V_1 0 0 1 ··· 0
$S_{64} + S_{65} + S_2 + \cdots + S_{127} = W_1 \times 256$
$S_0 + S_1 + S_{66} + \cdots + S_{63} = W_3 \times 256$

Figure 29: Mapping 4 weights on two rows in the crossbar.

It has been shown in previous studies that DNNs can be quantized to 8 bits without losing accuracy. Thus, SRA uses 256-bit streams in SC format for weights and inputs to preserve the same precision of 8-bit binary format. For a 256×256 crossbar and a GSL vector, there are 256 logical bit lines, each consisting of 256 bits. It is straightforward to map 256 weights onto the crossbar by storing one weight on each logical bit line. However, this incurs a 32× storage overhead compared to the original 8-bit binary format. In this section, we propose a novel bit stream mapping strategy to store more than 256 weights on one crossbar to reduce the storage overhead.

We use multiple GSL vectors. Each GSL vector maps a different set of 256 weights on the crossbar. For n GSL vectors, a crossbar can store $256n$ weights, with every pair of adjacent physical bit lines constructing $2n$ logical bit line to $2n$ weights. Figure 29 shows an example of mapping 4 weights on the first two physical bit lines in the crossbar assuming $n = 2$. Each square in the figure represents a segment. S_i indicates the number of 1s in each segment. Since a segment has 4 cells, S_i can only be 0, 1, 2, 3, or 4. V_0 and V_1 are the two GSL vectors. The first weight W_0 is mapped to the logical bit line determined by V_0, i.e, S_0, S_{65}, S_{66}, ..., S_{63}. The second weight W_1 is mapped to the logical bit line determined by V_1, i.e, S_{64}, S_{65}, S_2, ..., S_{127}. By using the negation of V_0 and V_1, we can get another two logical bit lines using the remaining segments to store W_2 and W_3. The left-hand side of the equations in Figure 29 represents the total number of 1s in each logical bit line. In order to correctly represent W_i, the total number of 1s in the logical bit lines needs to be $W_i \times 256$, which is the right-hand side of the equations. The mapping process is to determine the S_is to satisfy those equations. However, when n increases, it becomes hard to make all the equations to be true. We formulate this problem as a mixed-integer linear programming (MIP) model that minimizes the difference between the left-hand sides and the right-hand sides. In general, the MIP model for mapping $2n$ weights on the two adjacent physical bit lines can be written as follows:

Minimize

$$\sum_{i=0}^{2n} D_i$$

Subject to:

$$S_j \in \mathbb{Z} \qquad (0 \leq j \leq 127)$$

$$S_j \geq 0 \qquad (0 \leq j \leq 127)$$

$$S_j \leq 4 \qquad (0 \leq j \leq 127)$$

$$D_i \geq \left(\sum_{j=0}^{63} V_{i,j} \cdot S_j + \sum_{j=0}^{63} \overline{V_{i,j}} \cdot S_{j+64}\right) - W_i \times 256 \qquad (0 \leq i \leq n\text{-}1)$$

$$D_i \geq W_i \times 256 - \left(\sum_{j=0}^{63} V_{i,j} \cdot S_j + \sum_{j=0}^{63} \overline{V_{i,j}} \cdot S_{j+64}\right) \qquad (0 \leq i \leq n\text{-}1)$$

$$D_i \geq \left(\sum_{j=0}^{63} \overline{V_{i,j}} \cdot S_j + \sum_{j=0}^{63} V_{i,j} \cdot S_{j+64}\right) - W_i \times 256 \qquad (n \leq i \leq 2n\text{-}1)$$

$$D_i \geq W_i \times 256 - \left(\sum_{j=0}^{63} \overline{V_{i,j}} \cdot S_j + \sum_{j=0}^{63} V_{i,j} \cdot S_{j+64}\right) \qquad (n \leq i \leq 2n\text{-}1)$$

D_i represents the difference between the two sides of the equations. The first three constraints restrict the number of 1s in each segment to be integers between 0 and 4. Since linear programming can not use absolute values as constraints, two inequalities are used to represent the absolute difference. The next two constraints indicate the difference when mapping weights on logical bit lines determined by V_is. The last two constraints indicate the difference when mapping weights on logical bit lines determined by $\overline{V_i}$s.

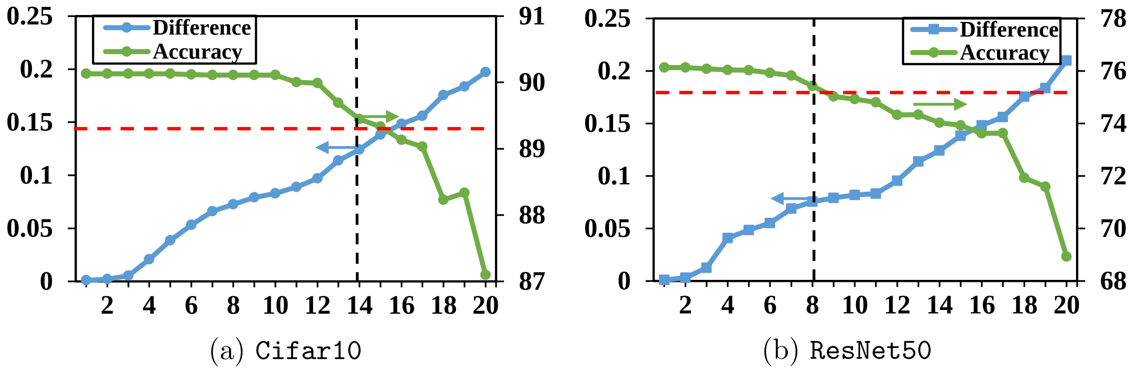


Figure 30: Tolerance of n for `Cifar10` and `ResNet50`.

When n becomes larger, it is harder to minimize the target. Figure 30 shows the mappings of `Cifar10` and `ResNet50` (structure details are listed in Table 10) when n increases. The blue lines illustrate the average difference between the left-hand sides and right-hand sides after optimization, while the green lines show the inference accuracy. From the figure, we can see that different DNNs have different tolerance on n. If we set the accuracy loss budget to 1% (marked by the red dashed line), n can be set to 14 in `Cifar10` while n can not exceed 8 for `ResNet50` which is also the worst case in all our tested DNNs. It is possible to use larger ns for different DNNs to achieve more storage savings, however, we conservatively set n to 8 for all DNNs for simplicity in evaluation.

5.2.4 SC Conversion

Conventional SC conversion uses linear feedback shift registers (LFSRs) to generate random bit streams with a specific percentage of 1s. However, LFSR based SC conversion can only generate 1 bit at a time, introducing a long latency for each conversion. For faster SC conversion, We propose to reuse the sign bits of the weights as the source to generate random bit streams.

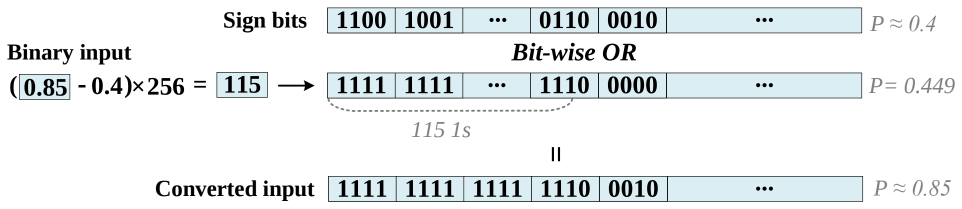


Figure 31: An example of converting 0.85 from binary format to SC format.

For each computation between the inputs and the weights, the weights' sign bits also need to be read out from the sign bit crossbar. Since the distribution of 1s in the sign bits of the weights is known beforehand, we can inject 0s or 1s into the sign bits to adjust the ratio of 1s according to the target input value. Figure 31 shows an example of converting the input 0.85 to SC format, assuming the ratio of 1s in the sign bits are 40% (represent a value of 0.4). In the example, 115 1s are injected into the sign bits consecutively starting from the first bit position. There may be other inject patterns by changing the start bit position of the consecutive 1s.

In order to protect the intermediate results, we use a hash table to select a different inject pattern for different inputs. Because 1s are not evenly distributed along the sign bits, using different inject patterns will introduce some random noises into the converted SC values. The hash table is kept confidential in an SRAM buffer on-chip and stored encrypted off-chip. Therefore, the adversary can only observe the binary format intermediate results stored off-chip while the SC format values participated in the computation are dynamically generated on-chip.

5.3 Accelerator Design

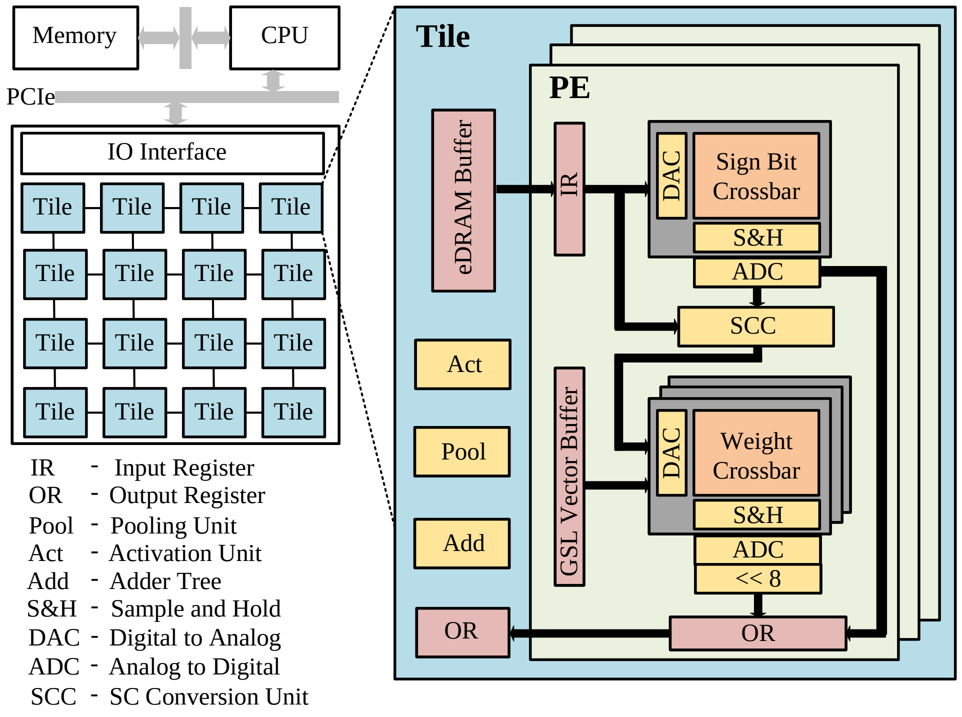


Figure 32: `SRA` accelerator.

Figure 32 shows the architecture of the proposed `SRA` accelerator, which is attached to the system through the PCIe bus and works as a slave to process DNN tasks received from the CPU. The `SRA` accelerator is a tiled structure that uses a concentrated mesh to provide communications between the tiles. The details of one tile are shown on the right of Figure 32. Each tile has an eDRAM buffer to store the binary format inputs of the DNN layer that is currently doing multiplications with the weights in the Processing Engines (PEs). An Output Register (OR) collects the PEs' multiplication results, which are then summed up in the adder tree. Each tile also has a Pooling unit for pooling layers and an Activation unit that implements the ReLU activation function. Each PE has a set of ReRAM crossbars. One of the crossbars stores the sign bits of the weights. The remaining crossbars are used for storing the SC format weights and performing multiplications with the inputs. Each PE also has an SRAM buffer to hold the *GSL* vectors. SRA uses 8 *GSL* vectors, each *GSL* vector has 64 bits. So, the size of the GSL vector buffer of each PE is 64B.

5.4 Experimental Methodology

Table 9: SRA accelerator parameters.

Unit	Spec	Power
PE		
ADC	num: 8; resolution: 8bits	16mW
DAC	num: 8×256; resolution: 1bit	8mW
S+H	num: 8×256	20uW
Xbar array	num: 8; size: 256×256; cell bits: 1	2.7mW
IR	size: 2KB	1.24mW
OR	size: 256B	0.23mW
GSL buffer	size: 64B	0.112mW
Shifters	size: 256	10uW
SCC	num: 1	0.02mW
PE Total	num: 1	28.4mW
Tile		
PE	num: 12	339.9mW
eDRAM	size: 64KB; banks: 2; width: 256	20.7mW
Bus	wires: 384	7mW
Router	flit size: 32; ports: 8	42mW
Activation	num: 2	0.52mW
Add	num: 1	0.05mW
Maxpool	num: 1	0.4mW
OR	size: 3KB	1.68mW
Tile Total	num: 1	412.3mW
Chip		
Tile	num: 168	69.2W
Hyper Tr	links: 4; freq: 1.6GHz	10.4W
Chip Total	num: 1	79.7W

We compare SRA with ISAAC [64] as it is the most widely used baseline in other ReRAM based DNN accelerator designs. Comparing with ISAAC makes it easy to scale SRA's performance and energy consumption numbers to compare with other works. Table 9 lists the detailed parameters of an SRA chip. Most of the configurations are kept the same with ISAAC, except that we use 8-bit quantized pre-trained weights, 256×256 crossbars, and single-bit ReRAM cells in SRA. To get a fair comparison, we make the same changes to ISAAC as the baseline during evaluation. We follow the equation in [63] to scale the ADC power consumption for a different ADC resolution. The power parameters of SRAM registers and eDRAM buffers are obtained by simulation using CACTI [55] at 32nm process assumed in ISAAC. We use NVSim [22] to get parameters of ReRAM crossbars. We evaluate SRA's performance and energy consumption by using a custom cycle-accurate simulator to simulate DNN's layer-by-layer execution on SRA. The simulated accelerator runs at 1.2GHz.

Table 10: Benchmarks.

Dataset	**Network**
MNIST	LeNet-5
SVHN	conv3x32-conv3x32-pool-conv3x64-conv3x64-pool-conv3x128-conv3x128-pool-1024-512-10
Cifar10	conv3x128-conv3x128-conv3x128-pool-conv3x256-conv3x256-conv3x256-pool-conv3x512-conv3x512-conv3x512-pool-1024-10
ImageNet	ResNet50, ResNeXt50, GoogLeNet, DenseNet

We test SRA on four datasets: MNIST [44], SVHN [56], Cifar10 [41] and ImageNet [21]. For SVHN and Cifar10, we construct custom neural network structures, whose details are listed in Table 10. We use LeNet-5 [45] for MNIST. We test ImageNet on four networks — ResNet50 [31], ResNeXt50 [77], GoogLeNet [69] and DenseNet [34]. All these four networks are popular ones in the machine learning community. We use PyTorch for fine-tuning and accuracy evaluation. we use the PuLP library to solve the MIP optimizations.

5.5 Results And Analysis

5.5.1 Protection

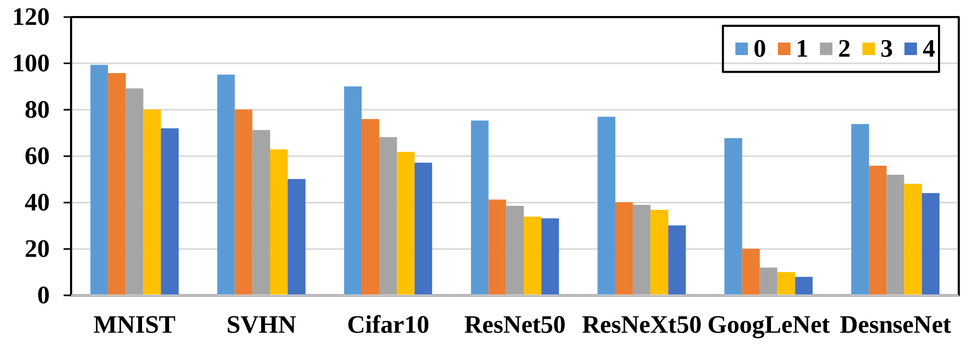


Figure 33: Accuracy with increasing number of *GSL* vectors.

Because the *GSL* vectors are not accessible to the attacker, even if the entire ReRAM content is known, the exact bit streams used in the computation are still hidden from the attacker. Each *GSL* vector has 64bits, the attacker needs to try 2^{64} different combinations to ensure the right one is included. In addition, there are $n = 8$ *GSL* vectors, so the total compute complexity is 2^{67}. Figure 33 shows the accuracy when the attacker guesses a *GSL* vector that is very close to the real *GSL* vector. The first bar is the baseline accuracy when the real *GSL* vector is used. The second bar shows the accuracy if there is only a 1-bit difference between the attacker guessed *GSL* vector and the real *GSL* vector. All of the benchmarks suffer a large accuracy loss. Note that `MNIST` is a very small dataset, it could be very easily trained to 99.9% accuracy. So a 5% accuracy loss shown in Figure 33 for `MNIST` already renders significant accuracy degradation. When more bits are guessed incorrectly, the accuracy degradation becomes even larger.

Table 11 shows the effectiveness of `SRA`'s protection on intermediate results. Assuming the target device uses inject pattern A for SC conversion. As can be seen from the table, the accuracy is very close to the baseline (less than 1% accuracy loss). If the DNN is stolen and applied to a pirate device, the attacker can only use a different inject pattern (assuming inject pattern B). As shown by the fourth column in Table 11, the inference on the pirate device suffers a huge accuracy degradation. The last column shows that the stolen DNN is also unusable if the attacker uses the binary format weights directly for inference.

Table 11: Inference Accuracy

	Baseline	Inject Pattern A	Inject Pattern B	Binary Weights
MNIST	99.35%	99.36%	14.98%	77.83
SVHN	95.94%	95.23%	67.75%	62.03
Cifar10	90.13%	90.11%	39.47%	52.29
ResNet50	76.13%	75.43%	14.82%	20.8
ResNeXt50	77.61%	77.01%	13.9%	17.84
GoogLeNet	69.77%	67.82%	10.44%	11.81
DenseNet	74.434%	73.86%	20.44%	32.55

5.5.2 Performance

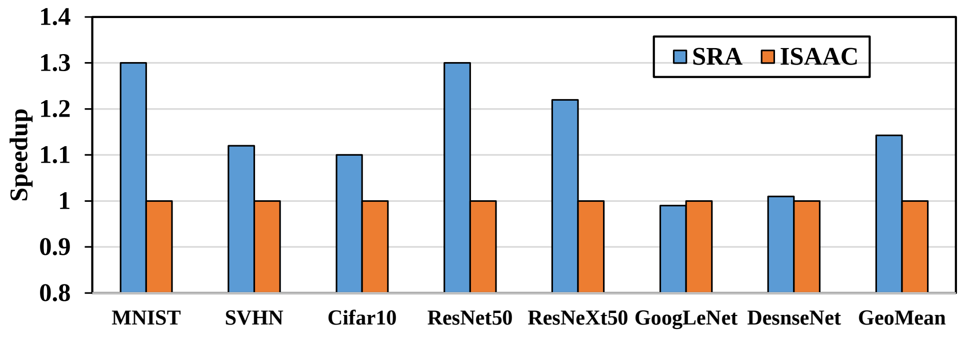


Figure 34: Speedup of SRA over ISAAC.

Figure 34 shows the performance of SRA compared to ISAAC. Even if SRA uses multiple *GSL* vectors to reduce the storage overhead caused by the long bit streams in SC format, the total weight storage is still 4× larger than the binary format in the baseline. So more cells need to be activated to perform computation. However, since SRA only activates half of the crossbar at a time, the read speed is much faster than activating the entire crossbar [78]. Also, because every segment of 4 cells is completely isolated from other cells in the crossbar, the sneak current problem is effectively suppressed, reducing the sensing delay of the ADCs. As a result, the overall latency is slightly better than the baseline. On average, there is a 1.14× speedup.

5.5.3 Energy

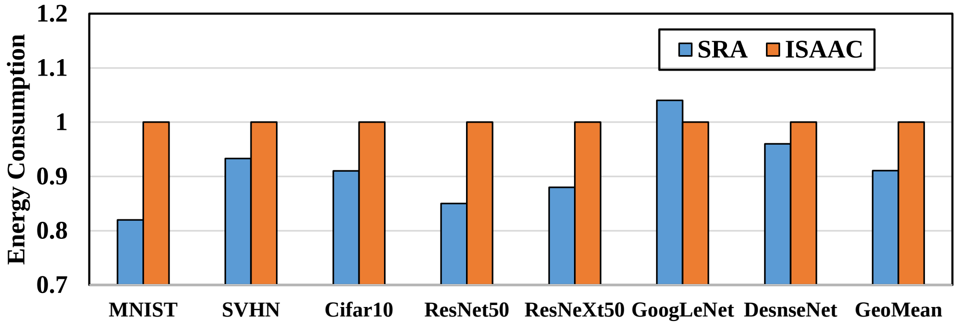


Figure 35: Energy consumption of SRA over ISAAC.

Figure 35 shows the energy consumption of running the tested benchmarks on SRA normalized to ISAAC. Although each computation needs two ReRAM accesses — one for reading the sign bits and the other for MAC operations, the overall energy consumption is still almost the same or slightly better than the baseline. This is because the SC conversion only reads one row out from the crossbar, its energy is much smaller than a full crossbar read. For MAC operation, SRA consumes less energy because only half of the cells are activated during computation, and the other half is completely shut off by the transistors of each segment. On average, SRA has 9% less energy consumption compared to ISAAC on average.

5.5.4 Comparing with SFGE

As introduced in Section 2.5.3, SFGE [10] is a recent work that proposes a sparse encryption scheme on weights stored in ReRAM crossbars. Because ReRAM cannot compute on encrypted data, SFGE needs to decrypt the weights before computing and encrypt the weights again when the computation finishes. In order to reduce this vulnerability time window, SFGE proposes a layer-by-layer execution fashion, i.e., at any given time there is only one layer's weights are in plaintext format. However, this still gives the adversaries the possibility to steal parts of the weights. One simple attack is to interrupt the execution (e.g., shut down the power and physically probe the memory) multiple times, such that eventually all layer's weights will be extracted. On the contrary, SRA dynamically reorganizes the segments to form logical bitlines. The plaintext weights are only transiently present by providing the GSL vectors. Therefore, there is no vulnerability time window in SRA.

Another issue with SFGE is the frequent writing to ReRAM cells. As every decryption and encryption needs to program the cells, ReRAM's low endurance of ReRAM limits the accelerator's lifetime. In addition, programming ReRAM cells also incurs energy and performance overhead. SRA does not need to write ReRAM cells after the weights are deployed. Therefore, SRA has benefits in energy consumption and performance, as well as no endurance concerns.

5.6 Conclusion

In this chapter, we introduced SRA — our third IP protection solution for DNN accelerators. It is also the first scheme that is for ReRAM based accelerators. Because ReRAM could retain data after the power is off, and it can only perform computation on plaintext data, existing encryption based schemes and our previous two schemes are no longer feasible. SRA exploits the 1T4R crossbar structure and the bit stream format in SC to hide the actual weight values stored in the crossbars, while still maintaining ReRAM's PIM capability. Like SCA, SRA supports protection in both training and inference phases, and provides protection for DNN weights and intermediate at the same time. Our experimental results show that SCA can effectively prevent pirating the deployed DNNs with negligible accuracy loss, and achieves 1.14× performance speedup and 9% energy reduction compared to ISAAC – a non-secure ReRAM-based baseline.

6.0 REDUCE SC STORAGE OVERHEAD IN RERAM CROSSBARS

In the previous chapter, we showed how `SRA` protects DNN weights while preserving SRAM crossbar's PIM capability. However, even `SRA` can store multiple SC format weight bit streams on the same pair of logical physical bit lines, it may still incur a higher storage overhead compared to the conventional binary format weights (ResNet50 has a 4$\times$ storager overhead according to Figure 30(b)). The fast growing size of DNNs in recent years can only make this problem even more severe. For example, AlexNet [42] — the first DNN winner of the ImageNet challenge in 2012 — only contains 60M parameters and 727M operations per input, VGG [66] won this challenge in 2014 with 136M parameters and 15B operations, and most recently one of the state-of-the-art DNNs EfficientNet-L2 [76] has 480M parameters and 612B operations. We can also see the increasing demand of computations as the DNN size grows. Although ReRAM provides much higher paralellism compared to CMOS based accelerators, it is legitimate to predict that future DNNs will also pose computation challenges on ReRAM if we cannot utilize its computation resource efficiently. In addition, because of ReRAM's low endurance and slow write speed, existing ReRAM based DNN accelerators require weights to be statically mapped on the crossbars to avoid frequently programming the cells. Therefore, it is essential to find reduce the storage overhead and improve the computation efficiency of ReRAM based DNN accelerators.

There have been many techniques that exploit weight sparsity and weight redundancy to reduce DNN size on CMOS based accelerators. Unfortunately, all these methods can not be effectively applied to ReRAM based DNN accelerators. Pruning [54] removes the values that are close to zero from the weight matrix. However, because ReRAM crossbars rely on regular matrix vector multiplication (MVM), exploiting the random and irregular distribution of zero weights requires complex control and peripheral circuits. Quantization [35] is another method that uses shorter bit width to represent weight values. However, quantization below eight bits usually incurs significant accuracy degradation [37, 40].

In this chapter, we propose `BFlip` to flip the bits in crossbars so that multiple bit matrices can share the same crossbar. Because the bits in a crossbar are accessed uniformly, separately flipping each bit induces large metadata and complex control logic. Therefore, we propose to only flip the bits in the granularity of rows and/or columns in a crossbar. We first cluster similar bit matrices together and then flip them to match the cluster's centroid bit matrix. For each cluster, only the centroid bit matrix is stored in the accelerator while all other bit matrices will be reconstructed from the centroid bit matrix during inference. However, since the bits can only be flipped in the granularity of rows and columns, it is very likely that a bit matrix cannot be flipped to perfectly match the centroid bit matrix. We apply a post-flipping calibration method that only updates the distribution statistics of batch normalization layers to mitigate the precision loss. Finally, We propose a ReRAM based accelerator that fully reaps the storage and computation benefits of `BFlip`.

6.1 Overview

The workflow of `BFlip` is illustrated in Figure 36. The whole workflow is composed of an offline phase to compress the DNN and an online phase to use the compressed DNN for inference. Given a pre-trained DNN, the *Decomposing* step decomposes each layer's weight matrix into bit matrices. In the *Clustering* step, the bit matrices are grouped into clusters. Each cluster has a centroid bit matrix computed by averaging all the bit matrices in the cluster. Within each cluster, the *Flipping* step finds a combination of row and column flips (referred to as *metadata*) for each bit matrix to make its distance to the centroid bit matrix as close as possible. Next, in the *Reconstructing* step, we reconstruct all the bit matrices by flipping the centroid bit matrix according to their metadata. In the *Calibrating* step, we re-run the forward propagation using the reconstructed bit matrices on the training dataset to mitigate the accuracy loss. If the calibration generates a satisfying accuracy, the centroid bit matrices and the metadata will be mapped to physical crossbars in the *Mapping* step and then enters the online phase for inference. If the accuracy is not satisfying, we first check whether it is still possible to improve the accuracy by applying more iterations in the *Flipping* step. If so, we re-execute the *Flipping* step with more iterations. Otherwise, we increase the number of clusters and go through the entire offline phase again until a satisfying accuracy is met.

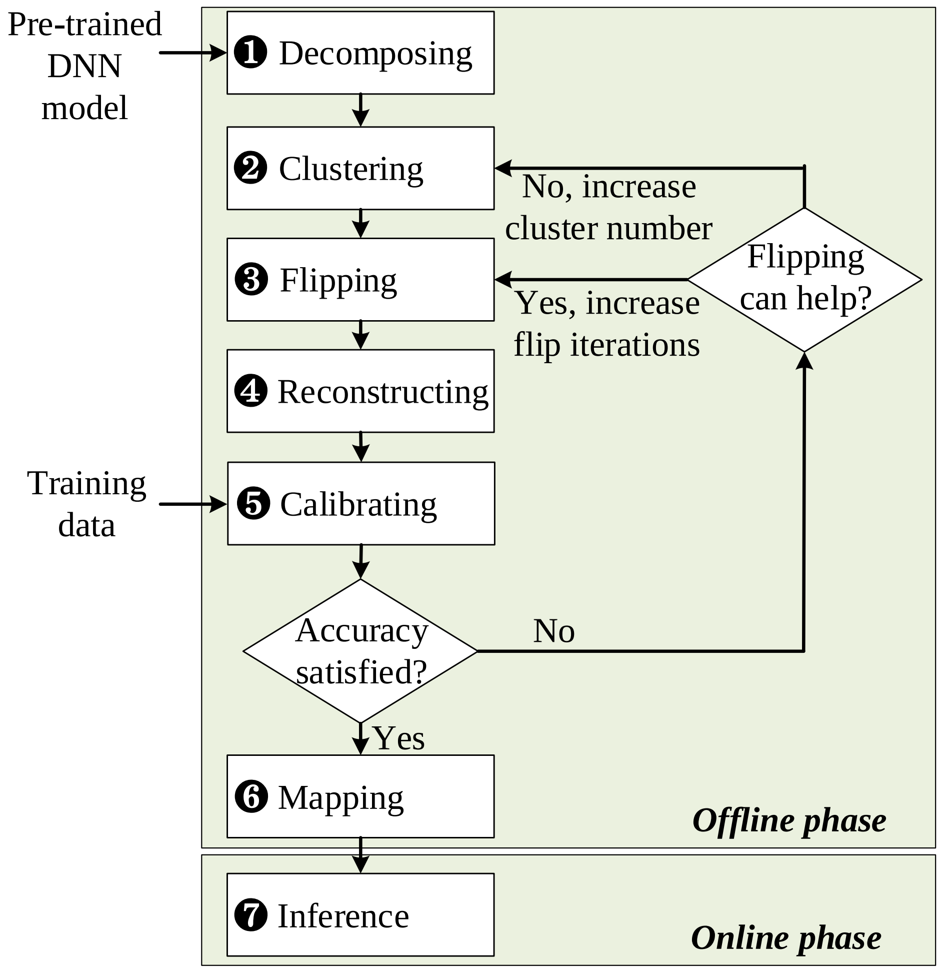


Figure 36: The overview of `BFlip`'s workflow.

6.2 Design Details

In this section, we describe the details of `BFlip` following the steps introduced in Section 6.1.

6.2.1 Decomposing Weight Matrices

Given a pre-trained DNN, the *Decomposing* step decomposes each layer's weight matrix into bit matrices. For example, if a weight matrix uses 8-bit fixed-point format to represent its weight values, the weight matrix is decomposed into eight bit matrices. Each bit matrix contains all the bits that have the same significance in the weight values. Then, the bit matrices are partitioned into segments of size $(n - 2m) \times (n - 2m)$. n is the crossbar size (i.e. n rows and n columns) and m is the maximum number of bit matrices that could share the same crossbar. These parameters are provided by the designer before the offline phase starts. All subsequent operations are performed on the partitioned bit matrices.

6.2.2 Clustering Bit Matrices

The *Clustering* step is to decide which bit matrices could share the same crossbar. We use Hamming Distance as the metric to measure the similarity between bit matrices. A small Hamming Distance between two bit matrices indicates that they are more likely to match each other after a small number of bit flips. We use Kmeans to group similar bit matrices into K clusters. K is a predefined parameter provided by the designer before the offline phase starts, but it may be changed in the following steps. Each cluster has a centroid bit matrix, such that the sum of the squared distances from all the bit matrices in the cluster to the centroid bit matrix is at the minimum. The centroid bit matrix is calculated by taking the average of all bit matrices in that cluster. The bit matrices that are in the same cluster will share the same crossbar. So K is limited by the available crossbars in the accelerator.

6.2.3 Flipping Bits in Bit Matrices

After clustering similar bit matrices, the next step is to minimize the distances from all the bit matrices in the cluster to the centroid bit matrix. A naive way is to find all the mismatched bits between each bit matrix and the centroid bit matrix, and only flip those bits to make them identical to the centroid bit matrix. However, this naive way will generate a large amount of metadata to record the positions of the flipped bits. Another problem is that MVM on a crossbar activates all the cells at the same time, flipping each bit individually breaks the regular access pattern and induces complex control overhead.

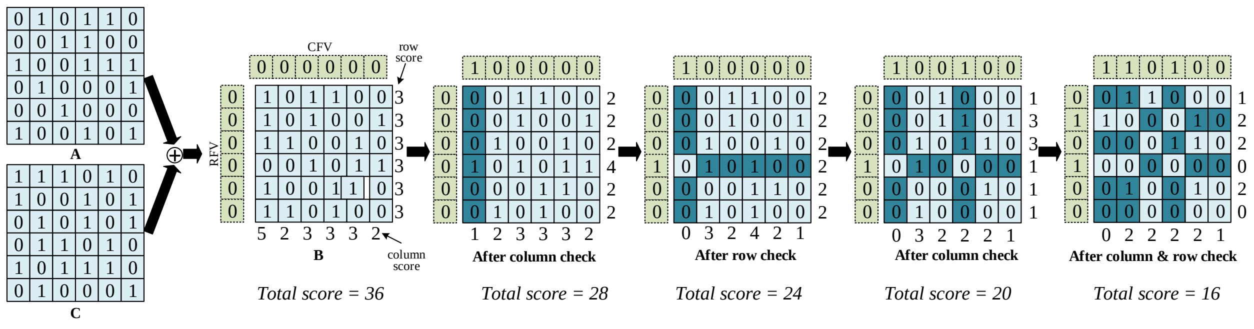


Figure 37: Example of flipping rows and columns to minimize two matrices' distance.

Therefore, we propose to flip the bits in the granularity of rows and columns. Each row and column only needs one bit to record whether it has been flipped or not. For a $t \times t$ bit matrix, there are only $2t$ bits metadata, which only has an overhead of $2/t$. We call the t-bit vector that records the flip status of each row the *row flip vector (RFV)*, and the other t-bit vector that records the flip status of each column the *column flip vector (CFV)*. To flip the bits in bit matrix A to match the centroid bit matrix C, we first calculate the difference between these two bit matrices by XORing them to get the bit matrix B, as shown in Figure 37. In B, 1 means there is a mismatch between the corresponding bits in A and C. We define a score variable s for each row and column in B, indicating the number of mismatched bits. When flipping the bits in a row or column, the corresponding score s changes to $t - s$. Finding the combination of row and column flips on A to match C, is equivalent to finding the combination of row and column flips on B to minimize its total score. Unfortunately, finding the optimal flip pattern can be reduced to the *Shortest Vector Problem (SVP)* in the lattice, which is known to be NP-hard [4]. Therefore, We propose a greedy approach to find a flip pattern that could make the two bit matrices as close as possible.

We first calculate all the column scores by counting the number of 1s in each column of B, if any of the scores is larger than $t/2$, then we flip this column and flip the bit in CFV

to record this column has been flipped. Then we calculate all the row scores and perform the same operation on the rows and RFV as we did for the columns. After the row check is done, it is possible some column scores that are previously less than $t/2$ now become larger than $t/2$. For example, the score of the forth column in Figure 37 turns from 3 to 4 after the row check. So we iteratively perform the column check and row check until all the scores are less than $t/2$.

Even though all the scores are less than $t/2$ after the column and row checks, the total score can still be reduced if one column and one row are flipped at the same time. For example, after the second column check in Figure 37, all the column scores and row scores are already less than $t/2$, and the total score is 20. If we flip the second column and the second row simultaneously, the total score can be further reduced to 16. Actually, in addition to flipping one column and one row simultaneously, we could also flip multiple columns and/or multiple rows simultaneously to further reduce the total score. In general, given a set of columns (denoted as C) and a set of rows (denoted as R), flipping all these columns and rows simultaneously could reduce the total score if the following equation is satisfied:

$$\sum_{c \in C} s_c + \sum_{r \in R} s_r > \frac{(|R| + |C|)t - 2t_0 + 2t_1}{2} \tag{6}$$

where s_c and s_r are the column score and row score respectively, $|C|$ is the number of columns in set C, $|R|$ is the number of rows in set R, t is the bit matrix size, t_0 is the number of intersection cells whose value is 0, t_1 is the number of intersection cells whose value is 1. The whole bit flipping algorithm is shown in Algorithm 1.

6.2.4 Reconstructing Bit Matrices

The metadata (CFVs and RFVs) produced in the previous step records the information of how to minimize the distance of each bit matrix in the cluster to the centroid bit matrix. So, we could either apply the flips on each bit matrix in the cluster to make it close to the centroid bit matrix, or we could apply the flips on the centroid bit matrix to make it more close to each bit matrix in the cluster. The *Reconstructing* step performs the latter to reconstruct each bit matrix in the cluster from the centroid bit matrix. Figure 38 shows

Algorithm 1 Bit Flipping

N_C: maximum number of columns that flip simultaneously

N_R: maximum number of rows that flip simultaneously

t: $t \times t$ bit matrix size

```
1:  $Flipped = True$
2:  while $Flipped$ do
3:      $Flipped = False$
4:      if Has columns whose score is larger than $t/2$ then
5:          $Flipped = True$
6:          Flip those columns
7:      end if
8:      if Has rows whose score is larger than $t/2$ then
9:          $Flipped = True$
10:         Flip those rows
11:     end if
12:     if $Flipped == False$ then
13:         for $i = 1$ to $N_C$, $j = 1$ to $N_R$ do
14:             Choose $i$ columns to form set $C$
15:             Choose $j$ rows to form set $R$
16:             if $C$ and $R$ satisfy eq. (6) then
17:                 $Flipped = True$
18:                 Flip these rows & columns simultaneously
19:             end if
20:         end for
21:     end if
22: end while
```

how to use the metadata generated in the previous step to reconstruct A from C. The reconstructed bit matrix is denoted as A'. The cells with dark background indicate the flipped cells. Because the *Flipping* step does not guarantee a flip pattern to make A and C identical, the reconstructed bit matrix A' is only an approximation of A. The mismatched cells between A and A' are marked with red numbers. In the same way, we could reconstruct an approximation of every original bit matrix in the cluster from the centroid bit matrix. As a result, we could get a new modified DNN consists of all the reconstructed bit matrices.

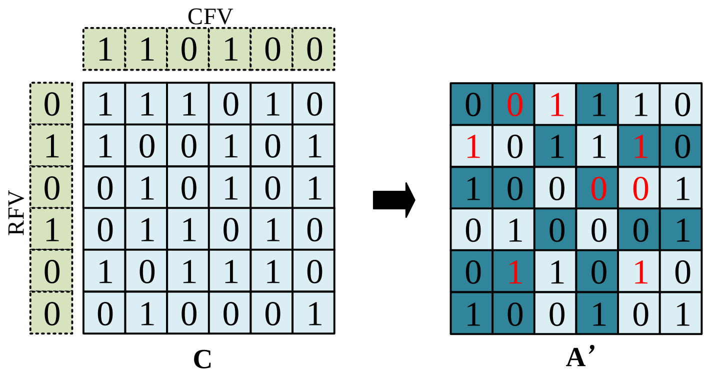


Figure 38: Reconstruct A' from C.

6.2.5 Calibrating the Modified DNN

Most recent DNNs contain batch normalization layers to accelerate the training process and improve the accuracy via a regularization effect. batch normalization layers standardize the inputs to DNN layers, i.e. making the each layer's inputs follow a standard distribution. However, calculating the actual mean and standard deviation of each layer's input during inference incurs a large computation overhead. So batch normalization layers use the statistics (mean and standard deviation) collected from the training data to standardize the inputs in the inference phase, based on the assumption that the training data has the same distribution of the whole data seen by the DNN in real applications. Because the modified

DNN produced in the previous step introduces noises to the weights which alters the distribution of the input to the next layer, the above assumption can no longer be held true. As shown in Table 12, there is an accuracy drop from 76.13% to 71.628% for `ResNet50` if we directly use the modified DNN in the inference phase. We tackle this problem by updating the distribution statistics in batch normalization layers. One thing to note here is that we only update the distribution statistics of batch normalization layers, instead of the trainable parameters (i.e. the gamma weights and beta weights) of the batch normalization layers. Because distribution statistics are collected during the forward propagation, we only need to re-run the forward propagation phase on the training data without the need for backward propagation and weight update. The last column in Table 12 shows the accuracy after updating the batch normalization statistics.

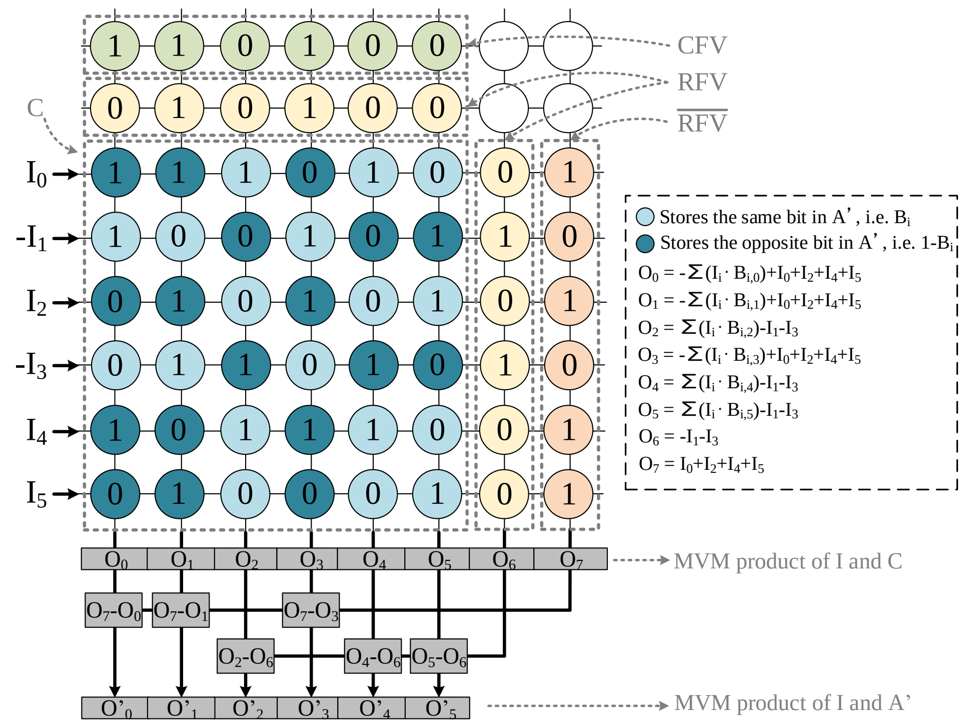
Table 12: Batch normalization's impact on accuracy.

Network	**baseline**	**no BN update**	**after BN update**
`ResNet50`	76.13%	71.628%	74.508%
`VGG16`	71.592%	58.214%	-
`VGG16-BN`	73.360%	68.966%	70.732%

For networks that do not have batch normalization layers, we add batch normalization layers after each convolutional layer and classification layer to mitigate the impact of weight noises introduced in previous steps. Table 12 also shows the accuracy of the `VGG16` network with and without batch normalization layers. Adding batch normalization layers not only increases the baseline accuracy, but also effectively mitigates the accuracy loss caused by mismatched bits introduced in previous steps.

6.2.6 Mapping to Crossbars

All the bit matrices in the same cluster can be constructed from the centroid bit matrix and the metadata. So they can share the same crossbar, and we only need to store the centroid bit matrix and the metadata. If a cluster has more than m matrices (m is the maximum number of matrices that can share the crossbar, defined in the *Decomposing* step), the cluster is divided into sub-clusters with size not greater than m. Each sub-cluster will map to a separate crossbar. In the *Decomposing* step, the bit matrices have been cut into sizes of $(n-2m)\times(n-2m)$, so the size of the centroid bit matrix is also $(n-2m)\times(n-2m)$. In addition to storing the centroid bit matrix on the crossbar, the remaining $2m$ rows and $2m$ columns of the crossbar store the metadata of each bit matrix in the sub-cluster. For each bit matrix, we use two rows to store its CFV and RFV, and two columns to store its RFV and the negation of RFV (i.e. $\overline{RFV}$). Figure 39 shows an example of mapping the centroid bit matrix C and the metadta of A in Figure 37 to a crossbar. If there are other bit matrices that also share this crossbar, each bit matrix needs two more rows to store its CFV and RFV and two more columns to store its RFV and $\overline{RFV}$.



CFV
RFV
$\overline{\text{RFV}}$
C
I_0
$-I_1$
I_2
$-I_3$
I_4
I_5
● Stores the same bit in A', i.e. B_i
● Stores the opposite bit in A', i.e. $1-B_i$
$O_0 = -\Sigma(I_i \cdot B_{i,0})+I_0+I_2+I_4+I_5$
$O_1 = -\Sigma(I_i \cdot B_{i,1})+I_0+I_2+I_4+I_5$
$O_2 = \Sigma(I_i \cdot B_{i,2})-I_1-I_3$
$O_3 = -\Sigma(I_i \cdot B_{i,3})+I_0+I_2+I_4+I_5$
$O_4 = \Sigma(I_i \cdot B_{i,4})-I_1-I_3$
$O_5 = \Sigma(I_i \cdot B_{i,5})-I_1-I_3$
$O_6 = -I_1-I_3$
$O_7 = I_0+I_2+I_4+I_5$
O_0 O_1 O_2 O_3 O_4 O_5 O_6 O_7
MVM product of I and C
O_7-O_0 O_7-O_1 O_7-O_3
O_2-O_6 O_4-O_6 O_5-O_6
O'_0 O'_1 O'_2 O'_3 O'_4 O'_5
MVM product of I and A'

Figure 39: Map centroid bit matrix and meta data to crossbar.

6.2.7 Inference

In Figure 39, because the crossbar only stores the centroid bit matrix, the output from the crossbar (i.e. O_0, O_1, ..., O_5) is the MVM product between input I and centroid bit matrix C. Note that when performing MVM in the crossbar, the top rows which store the metadata do not participate in the computation by applying a zero voltage on their wordlines. In order to get the MVM product between I and A', additional steps are required to adjust the O_is. The dark cells in Figure 39 indicates the different cells between C and A'. If we denote the bits in A' as B_i, the dark cells stores $1 - B_i$. If the corresponding bit in RFV is 1, we apply the opposite value of the input to on the wordline. As a result, for O_0, O_2, O_4, and O_5, whose bit in CFV is 1, the output is $-\sum(IB) + I_0 + I_2 + I_4 + I_5$. We can adjust these outputs by subtracting them from the output of O_7. And for O_1 and O_3, whose bit in CFV is 0, the output is $\sum(IB) - I_1 - I_3$. We can adjust these outputs by adding the output of O_6. The adjusted outputs (i.e. O'_0, O'_1, ..., O'_5) are the MVM product between I and A'.

6.3 Accelerator Design

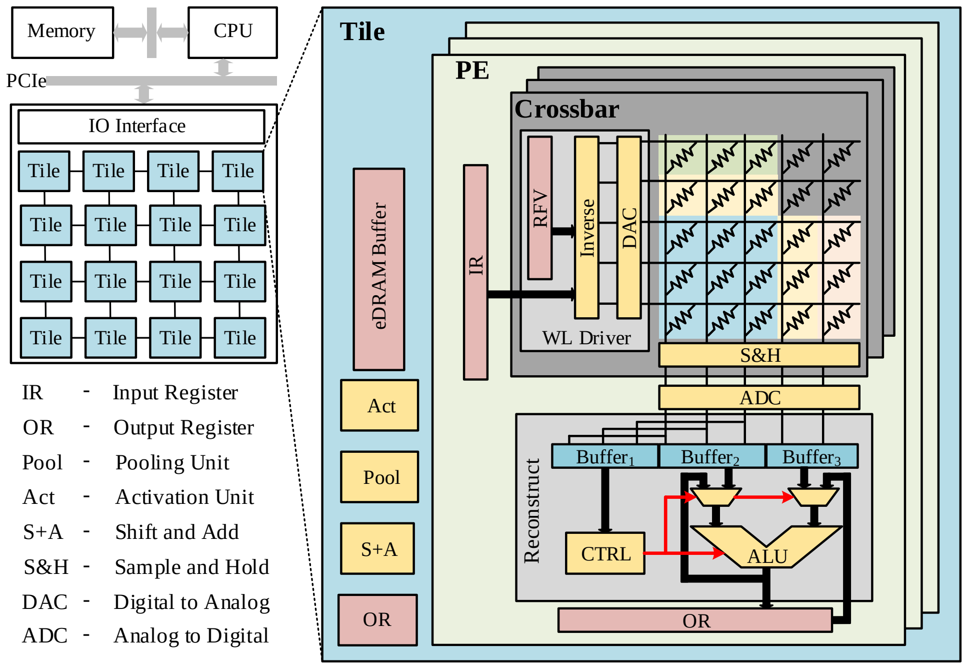


Figure 40: `BFlip` accelerator.

In this section, we present our `BFlip` accelerator. The accelerator is integrated into the system via PCIe bus and works as a slave to process DNN tasks received from the CPU. The top-level structure of the `BFlip` accelerator follows the generic ReRAM based DNN accelerator design, which consists of multiple tiles as shown in Figure 40. The tiles are connected using an on-chip concentrated mesh. The right of Figure 40 shows the details of one such tile. Each tile has an eDRAM buffer to store inputs of DNN layers, an output buffer for output aggregation, a Shift and Add unit to form the full production results from bit slices, a Pooling unit for pooling layers, and an Activation unit that implements non-linear activation functions. The computation is distributed to multiple Processing Engines (PEs). Each PE has a set of ReRAM crossbars to perform the MVM computation. Each crossbar is partitioned into four areas — the rows at the top store the CFVs (shown in green) and RFVs (shown in yellow), the right columns store the RFVs (shown in yellow) and $\overline{RFV}$s (shown in pink), and the left bottom part to store the centroid bit matrix (shown in blue).

Both the inputs and the weights are in two's complement format.

A MVM computation for one bit matrix consists of three steps. In the first step, the RFV vector and the CFV vector are read from the top rows from the crossbar. The CFV vector is stored in $Buffer_1$ of the Reconstruct unit. The RFV vector is loaded into the RFV buffer in the wordline driver. In the second step, the inputs are converted to their opposite values according to the bits in the RFV buffer. Then, the inputs compute with the centroid bit matrix and the right-most columns which store the RFVs and $\overline{RFV}$s. The results of the centroid bit matrix are stored in $Buffer_2$ of the Reconstruct unit. The results of the RFVs and $\overline{RFV}$s are stored in $Buffer_3$ of the Reconstruct unit. In the third step, the ALU adjusts the results in the three buffers to get the MVM product between the input and the reconstructed bit matrix.

6.4 Experimental Methodology

6.4.1 Accuracy Models

We evaluate four datasets: `MNIST` [44], `SVHN` [56], `Cifar10` [41] and `ImageNet` [21]. We construct custom neural network structures for `SVHN` and `Cifar10`. The details of these structures are listed in Table 13. For `MNIST`, we run it on `LeNet-5` [45]. For `ImageNet`, we test it on five different networks — `VGG16` [66], `ResNet50` [31], `ResNeXt50` [77], `GoogLeNet` [69] and `DenseNet` [34]. All these five networks are popular ones in the machine learning community.

Table 13: Benchmarks.

Dataset	**Network**
MNIST	LeNet-5
SVHN	conv3x32-conv3x32-pool-conv3x64-conv3x64-pool-conv3x128-conv3x128-pool-1024-512-10
Cifar10	conv3x128-conv3x128-conv3x128-pool-conv3x256-conv3x256-conv3x256-pool-conv3x512-conv3x512-conv3x512-pool-1024-10
ImageNet	VGG16, ResNet50, ResNeXt50, GoogLeNet, DenseNet

6.4.2 Hardware Models

We implement a custom cycle-accurate simulator for the BFlip accelerator. The hardware parameters are listed in Table 14. Most of the parameter values are adopted from ISAAC [64]. For components that are new in the BFlip accelerator, we use CACTI [55] and NVSIM [22] to model SRAM buffers and ReRAM crossbars, respectively. Both SRAM and logic use 32nm process. Crossbar arrays use single-bit ReRAM cells, which need 4.4ns for read and 52.2ns for write. The DAC's resolution is one bit and the ADC's resolution is seven bits. We use the same ADC design as in ISAAC, and scale the power consumption reported in ISAAC to seven bits using the equation in [63].

Table 14: `BFlip` accelerator parameters.

Unit	**Spec**	**Power**
Resconstruct Unit		
Buffer	size: 48B	0.07mW
ALU	num: 1	0.1mW
PE		
ADC	num: 8; resolution: 7bits	9mW
DAC	num: 8×128; resolution: 1bit	4mW
S+H	num: 8×128	10uW
Xbar array	num: 8; size: 128×128; cell bits: 1	1.8mW
RFV buffer	size: 16B	0.028mW
Reverse	num: 1	0.01mW
Reconstruct	num: 4	0.64mW
IR	size: 2KB	1.24mW
OR	size: 256B	0.23mW
PE Total	num: 1	17
Tile		
PE	num: 12	204mW
eDRAM	size: 64KB; banks: 2; width: 256	20.7mW
Bus	wires: 384	7mW
Router	flit size: 32; ports: 8	42mW
Activation	num: 2	0.52mW
S+A	num: 1	0.05mW
Maxpool	num: 1	0.4mW
OR	size: 3KB	1.68mW
Tile Total	num: 1	276mW
Chip		
Tile	num: 168	46.3W
Hyper Tr	links: 4; freq: 1.6GHz	10.4W
Chip Total	num: 1	56.7W

We compare BFlip's performance and energy consumption with two existing works — ISAAC [64] and SRE [78]. ISAAC is the most widely used baseline in ReRAM based DNN accelerator designs. The performance and energy comparison results with ISAAC can be conveniently scaled to compare with other ReRAM based DNN accelerators. SRE is one of the state-of-the-arts that exploit DNN weight sparsity in ReRAM based DNN accelerators. Both ISAAC and SRE use two-bit cells. We modify these two baselines to use single-cells for a fair comparison with BFlip.

6.5 Results And Analysis

In this section, we divide the evaluation of BFlip into two parts. In the first part, we first evaluate BFlip's impact on accuracy with different ms (i.e., the number of bit matrices that share the same crossbar). In terms of compression ratio, changing m in BFlip has the same effect as using different bit widths in quantization. Table 15 shows the equivalent quantized bit width for different ms, assuming crossbar size is 128×128. In the second part, we analyze BFlip's benefits of performance improvement and energy reduction.

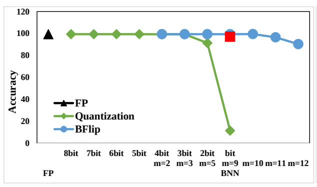
Table 15: Equivalent quantized bit width for different m.

Quant. bit width	4	3	2	1
m for BFlip	2	3	5	9

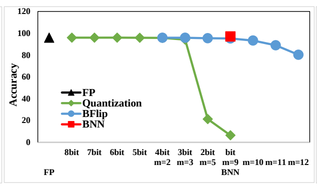
6.5.1 Accuracy

Figure 41 shows the accuracy results of BFlip, the full-precision baseline, and the conventional quantization method. The full-precision baseline is obtained from the PyTorch torchvision package. For quantization, eight bits are sufficient for all benchmarks to keep the accuracy the same as the full-precision baseline. However, further reducing the bit width will significantly degrade the accuracy. Small datasets can sustain a more aggressive quantization. For example, there is an accuracy loss of only 0.09%, 0.16%, and 0.3% for

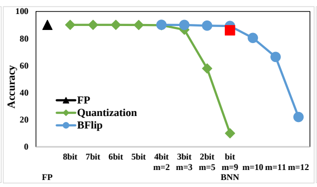
MNIST, SVHN, and Cifar10 when their weights are quantized to four bits. However, DNNs for ImageNet need at least seven bits to prevent a massive loss of accuracy. On the other hand, we observe little accuracy loss for BFlip even when $m = 9$ (equivalent to 1-bit quantization). Binarized neural networks (BNNs) have gained a lot of attention as each weight value only needs one bit. BNNs are different from convention 1-bit quantization in that many optimizations are made in either the quantization method or the training process. The accuracy gap between BNNs and the full-precision DNN is getting smaller in recent years. Figure 41 also shows the accuracy of recent state-of-the-art results of BNNs for each benchmark. For ImageNet, there is still a 6.26% to 22.6% accuracy gap between BNN and the full-precision baseline. However, the accuracy loss of BFlip with $m = 9$ (equivalent to 1-bit quantization) is still negligible (2.18% on average).



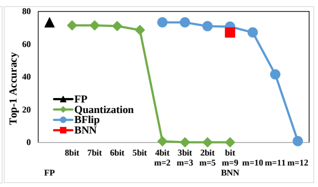
(a) MNIST



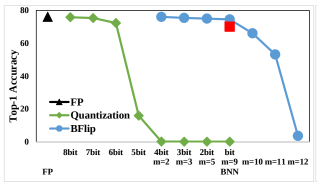
(b) SVHN



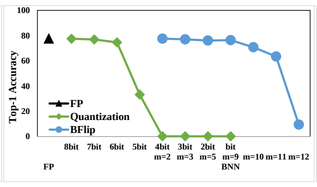
(c) Cifar10



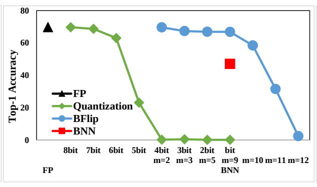
(d) VGG16



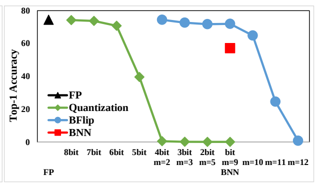
(e) ResNet50



(f) ResNeXt50



(g) GoogLeNet



(h) DenseNet

Figure 41: Accuracy of the full-precision baseline, quantization, BNN and BFlip.

6.5.2 Performance and Energy

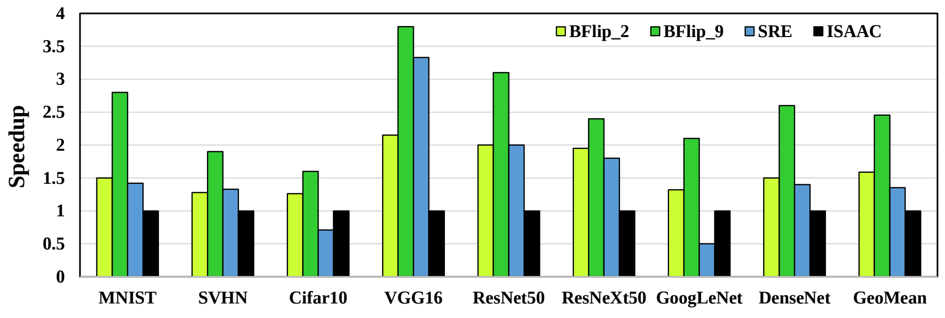


Figure 42: Speedup of `BFlip` over baselines.

We select $m = 2$ and 9 for `BFlip` as examples to compare the performance speedup with the baselines. We select these two configurations because $m = 2$ is equivalent to quantizing weight bit width to four bits and the accuracy loss is negligible for small datasets, while $m = 9$ is equivalent to BNNs which have been wildly studied in previous works. Figure 42 shows the performance speedup of `BFlip` over the baselines. All the results are normalized to `ISAAC`. `BFlip`'s performance speedup ranges from 1.26× to 2.15× for $m = 2$ and 1.6× to 3.8× for $m = 9$, and the average speedup is 1.58× and 2.45×, respectively. The performance speedup comes from reusing the crossbar outputs, as the crossbar based MVM is the most time-consuming operation. Using larger m also helps to reduce the computation bottleneck, because only a smaller region inside the crossbar needs to be accessed simultaneously for computation, which needs significantly less access time than activating the whole crossbar. For example, when $m = 9$, one crossbar computation result can be used to generate the MVM product of nine bit matrices, and only a 110×128 sub-region inside the crossbar needs to participate in the MVM computation. Because the computation saving in `SRE` depends on the sparsity of the pruned weights, its performance is worse than `ISAAC` for dense DNNs (`Cifar10` and `GoogLeNet`).

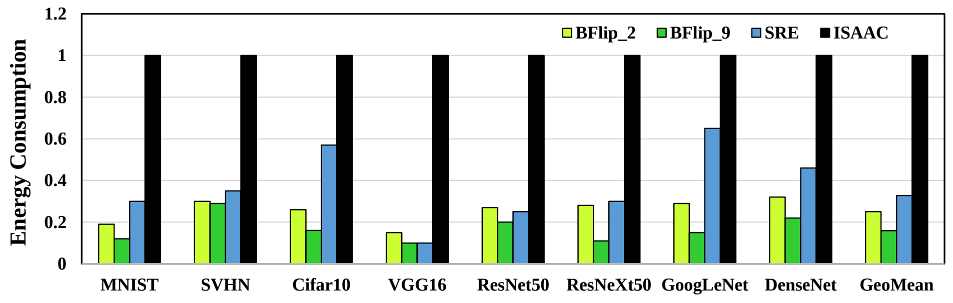


Figure 43: Energy consumption of `BFlip` over baselines.

Figure 43 shows the energy consumption of `BFlip` when $m = 2$ and 9 normalized to `ISAAC`. `SRE` reduces energy consumption by skipping all-zero rows in OUs (Operation Unit) — a fine-grain sub-region inside a crossbar. However, because the inputs need to be reordered due to the irregular access pattern in the crossbar, it demands additional eDRAM accesses which incur a large energy overhead. So, it can save more energy consumption on DNNs with structural sparsity (for example, 90% energy save for `VGG16`). `BFlip` only needs to convert the inputs into their opposite values, the order is not changed, so no additional eDRAM access is required. The energy saving of `BFlip` does not depend on sparsity structure after pruning, it saves 75% and 85% energy on average for $m = 2$ and 9, respectively.

6.6 Conclusion

In the this chapter, we introduced `BFlip` — a novel DNN compression scheme that works in ReRAM based crossbars to further reduce `SRA`'s storage overhead induced by the long bit streams in SC. `BFlip` clusters similar bit matrices together, and finds a combination of row and column flips for each bit matrix to minimize its distance to the centroid of the cluster. Therefore, only the centroid bit matrix is stored in the crossbar, which is shared by all other bit matrices in that cluster. We also proposed a calibration method to improve the accuracy as well as a ReRAM based DNN accelerator to fully reap the storage and computation benefits of `BFlip`. `BFlip` is not only application to `SRA`, but also is a general DNN compression method for non-SC based ReRAM accelerators. Our experimental results show that `BFlip` effectively reduces DNN's model size and computation with negligible accuracy impact. The proposed `BFlip` accelerator achieves 2.4× speedup and 85% energy reduction over the `ISAAC` baseline.

7.0 CONCLUSION AND FUTURE WORK

7.1 Summary

Training DNNs involves tremendous efforts in training their weights. Thus, it is critical to guarantee the security of the DNN IPs. However, traditional encryption based protection methods introduce additional computation overheads. As a result, this thesis initiates and explores a different line of IP protection solutions.

As DNNs are error tolerant algorithms, and there exist multiple solutions for a DNN to solve a specific task, we intentionally inject error bits into DNN weights during training. The error bits will be stuck at 0 or 1 during the entire training phase and remains the stuck bits in the trained weights. By making the distribution of the error bits correlated with the memory intrinsic errors caused by reducing DRAM timing, the trained weights are device dependent and will only produce accurate results on this specific device. In this way, not only encryption can be avoided, but also the computation performance will be improved thanks to the reduced DRAM timing.

To extend the protection capability to accelerators that support both training and inference, we used the precision errors in the same error tolerant training as the previous design. The precision errors are introduced during the conversion of weights from binary format to SC format. We modified the conversion method to use eDRAM errors to generate the random bit streams, thus the precision error distribution is also device dependent. Whenever the weights and intermediate results are off-loaded outside of the accelerator chip, they will be converted into binary format. Thanks to the precision difference, adversaries will never know the actual SC format weight values participated in computing.

ReRAM poses new challenges to DNN IP protection and there yet are no solutions for DNN protection on ReRAM based accelerators. We employ the 1T4R crossbar structure to make the bitline divided into small slices. Weights are in SC format, and each bit stream is scattered on multiple slices. Without knowing the scatter pattern, there is no way for the adversaries to know any entire bit stream. While this scheme can effectively protect the weight values, the long bit streams induce a large storage overhead on the crossbars. We also proposed a slice sharing scheme to reduce the total number of crossbars required to store DNN weights. As model compression is a hard topic for ReRAM accelerators, our slice sharing scheme fills the gap between effectiveness of model compression and ReRAM's tightly coupled crossbar structure requirement.

Even after the slice sharing is applied, the previous design still incurs a 4 times storage overhead compared to conventional non-secure ReRAM designs. We introduced another bit level sharing scheme to further reduce the model size on ReRAM crossbars. While the previous model compression technique is for SC domain computation specifically, this scheme can also be applied in conventional ReRAM PIM accelerators that are in binary format.

7.2 Future Research Directions

Whereas the designs in this thesis provide a variety of protection schemes for DNN weights with different security guarantees, there are still other security aspects that worth deeper exploration. Back to the categories we defined introducing related works (Section 2.5), this section describes the possible future research directions in the other two categories.

7.2.1 Algorithm Validity

Adversarial attacks are the most widely studied aspect of DNN security. Although most of the defense methods are algorithm based, there are a few attempts to utilize hardware support to defend against adversarial attacks. However, these protection approaches introduce additional hardware overhead. For example, DNNGuard [74] integrate a CPU core in the DNN accelerator for adversarial sample detection. 2-in-1 [24] needs special hardware to dynamically change the weight precision. Therefore, it is highly desirable to design an accelerator that achieves both defense effectiveness and computing efficiency.

Recently, there have been proposed works that leverage randomness for defending against adversarial attacks. Wang *et al.* [73] found enabling dropout during inference can significantly reduce the attack success rate. The dynamically changed weight precision in [24] is based on a similar observation. Inspiring from these works, we may introduce randomness from our memory timing errors that are generated in the same way as in `AEP` or `SCA`. In this way, it is possible to design accelerators that could defend against both adversarial attacks and IP attacks.

7.2.2 User Privacy

User privacy is becoming an important topic in DNN computing, especially in the setting of machine learning as a service (MLaaS) in the cloud. Existing solutions include homomorphic encryption (HE), secure multi-party computation (SMC), Garbled Circuits (GC), or a combination of these techniques. Although there are different approaches to solving the privacy issue, each of them has its own limitations. SMC and GC protect computing data by distributing different shares of the operand to different parties, while each party cannot infer the original data separately. The bottleneck is to distribute the shares to each party and exchange the shared information between parties. In the architecture research community, HE is the most widely explored approach to providing privacy protections. HE does not need communications between different parties during computing, but it introduces orders of magnitude overhead of both computation and storage. For example, the first attempt to integrate HE in DNN inference CryptoNets [25] has a 30000 $\times$ latency overhead and requires 491520 $\times$ more space to store the encrypted data compared to the unencrypted version if the batch size is 1, which is the common case in inference. Although subsequent works made many improvements on HE based DNN computing [**?**, 8], there is still thousands of times gap in both storage and latency. Even with the hardware support [60, 23, 28], latency is still not comparable to the unencrypted version. Moreover, the energy efficiency is much worse in these accelerators. To prevent the overhead from going too high, HE based approaches usually only encrypt one of the two operands while leaving the other in plaintext format. Thus, they can only protect either private inputs or DNN weights, but not both. One possible solution is to implement HE acceleration on top of `AEP`. The inputs are encrypted using HE, while the DNN weights are first converted into HE plaintext format and then use memory errors to protect them. This could not only provide full protection on both private inputs and DNN weights but also reduces energy consumption.

Bibliography

[1] *Global Standards for the Microelectronics Industry.* https://www.jedec.org/.

[2] Aditya Agrawal, Amin Ansari, and Josep Torrellas. *Mosaic: Exploiting the spatial locality of process variation to reduce refresh energy in on-chip eDRAM modules.* International Symposium on High Performance Computer Architecture (HPCA), 2014.

[3] Nabihah Ahmad and SM Rezaul Hasan. *Low-power compact composite field AES S-Box/Inv S-Box design in 65 nm CMOS using novel XOR gate.* Integration, the VLSI Journal, 2013.

[4] Miklós Ajtai. *The shortest vector problem in L2 is NP-hard for randomized reductions.* Symposium on Theory of Computing (STOC), 1998.

[5] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. *Theano: A Python framework for fast computation of mathematical expressions.* arXiv, 2016.

[6] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. *DRAM refresh mechanisms, penalties, and trade-offs.* IEEE Transactions on Computers, 2015.

[7] Mahdi Nazm Bojnordi and Engin Ipek. *Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning.* International Symposium on High Performance Computer Architecture (HPCA), 2016.

[8] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. *Low latency privacy preserving inference.* International Conference on Machine Learning (ICML), 2019.

[9] Geoffrey W Burr, Robert M Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U Giacometti, et al. *Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element.* IEEE Transactions on Electron Devices, 2015.

[10] Yi Cai, Xiaoming Chen, Lu Tian, Yu Wang, and Huazhong Yang. *Enabling Secure in-Memory Neural Network Computing by Sparse Fast Gradient Encryption.* International Conference on Computer-Aided Design (ICCAD), 2019.

[11] Nicholas Carlini and David Wagner. *Towards evaluating the robustness of neural networks.* International Conference on Learning Representations (ICLR), 2022.

[12] Cangxiong Chen and Neill DF Campbell. *Understanding Training-Data Leakage from Gradients in Neural Networks for Image Classification.* NeurIPS 21 Workshop 'Privacy in Machine Learning, 2021.

[13] Huili Chen, Bita Darvish Rohani, and Farinaz Koushanfar. *Deepmarks: A digital fingerprinting framework for deep neural networks.* International Conference on Multimedia Retrieval (ICMR), 2019.

[14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. *Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. *Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks.* International Symposium on Computer Architecture (ISCA), 2016.

[16] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. *Dadiannao: A machine-learning supercomputer.* International Symposium on Microarchitecture (MICRO), 2014.

[17] Todd Alan Christensen and II John Edwards Sheets. *Implementing physically unclonable function (PUF) utilizing EDRAM memory cell capacitance variation.* US Patent 8,300,450, 2012.

[18] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. *Deep learning with COTS HPC systems.* International Conference on Machine Learning (ICML), 2013.

[19] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *Training deep neural networks with low precision multiplications.* arXiv preprint arXiv:1412.7024, 2014.

[20] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *Binaryconnect: Training deep neural networks with binary weights during propagations.* Annual Conference on Neural Information Processing Systems (NeurIPS), 2015.

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. *Imagenet: A large-scale hierarchical image database.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.

[22] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. *Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2012.

[23] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Chris Peikert, and Daniel Sanchez. *F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption.* International Symposium on Microarchitecture (MICRO), 2021.

[24] Yonggan Fu, Yang Zhao, Qixuan Yu, Chaojian Li, and Yingyanel Lin. *2-in-1 Accelerator: Enabling Random Precision Switch for Winning Both Adversarial Robustness and Efficiency.* International Symposium on Microarchitecture (MICRO), 2021.

[25] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. *Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy.* International Conference on Machine Learning (ICML), 2016.

[26] Amira Guesmi, Ihsen Alouani, Khaled N Khasawneh, Mouna Baklouti, Tarek Frikha, Mohamed Abid, and Nael Abu-Ghazaleh. *Defensive approximation: securing CNNs using approximate computing.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021.

[27] Jia Guo and Miodrag Potkonjak. *Watermarking deep neural networks for embedded systems.* International Conference on Computer-Aided Design (ICCAD), 2018.

[28] Saransh Gupta and Tajana Šimunić Rosing. *Accelerating Fully Homomorphic Encryption with Processing in Memory.* Design Automation Conference (DAC), 2021.

[29] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. *Deep learning with limited numerical precision.* International Conference on Machine Learning (ICML), 2015.

[30] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. *DarKnight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware.* International Symposium on Microarchitecture (MICRO), 2021.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep residual learning for image recognition.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[32] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. *Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups.* IEEE Signal processing magazine, 2012.

[33] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. *Deepsniffer: A dnn model extraction framework based on learning architectural hints.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.

[34] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. *Densely connected convolutional networks.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[35] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. *Quantized neural networks: Training neural networks with low precision weights and activations.* The Journal of Machine Learning Research (JMLR), 2017.

[36] Sergey Ioffe and Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* International Conference on Machine Learning (ICML), 2015.

[37] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. *Quantization and training of neural networks for efficient integer-arithmetic-only inference.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

[38] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. *In-datacenter performance analysis of a tensor processing unit.* International Symposium on Computer Architecture (ISCA), 2017.

[39] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. *Reduced-precision strategies for bounded memory in deep neural nets.* arXiv preprint arXiv:1511.05236, 2015.

[40] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper.* arXiv preprint arXiv:1806.08342, 2018.

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

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. *Imagenet classification with deep convolutional neural networks.* Neural Information Processing Systems (NeurIP), 2012.

[43] Quoc V Le. *Building high-level features using large scale unsupervised learning.* IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013.

[44] Yann LeCun. *The MNIST database of handwritten digits.* http://yann.lecun.com/exdb/mnist/, 1998.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 1998.

[46] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan, Vivek Seshadri, Kevin Chang, and Onur Mutlu. *Adaptive-latency DRAM: Optimizing DRAM timing for the common-case.* International Symposium on High Performance Computer Architecture (HPCA), 2015.

[47] Jong Chern Lee, Jihwan Kim, Kyung Whan Kim, Young Jun Ku, Dae Suk Kim, Chunseok Jeong, Tae Sik Yun, Hongjung Kim, Ho Sung Cho, Sangmuk Oh, et al. *High bandwidth memory (HBM) with TSV technique.* International SoC Design Conference (ISOCC), 2016.

[48] Shuangchen Li, Alvin Oliver Glova, Xing Hu, Peng Gu, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. *Scope: A stochastic computing engine for dram-based in-situ accelerator.* International Symposium on Microarchitecture (MICRO), 2018.

[49] Wen Li, Ying Wang, Huawei Li, and Xiaowei Li. *P3M: a PIM-based neural network model protection scheme for deep learning accelerator.* Asia and South Pacific Design Automation Conference (ASPDAC), 2019.

[50] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. *Architectural support for copy and tamper resistant software.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[51] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn. *Flikker: Saving DRAM refresh-power through critical data partitioning.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011.

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

[53] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. *Shredder: Learning noise distributions to protect inference privacy.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.

[54] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. *Pruning convolutional neural networks for resource efficient inference.* International Conference on Learning Representations (ICLR), 2017.

[55] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. *CACTI 6.0: A tool to model large caches.* HP laboratories, 2009.

[56] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. *Reading digits in natural images with unsupervised feature learning.* NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.

[57] Karl Ni, Roger Pearce, Kofi Boakye, Brian Van Essen, Damian Borth, Barry Chen, and Eric Wang. *Large-scale deep learning on the YFCC100M dataset.* Annual Conference on Neural Information Processing Systems (NeurIPS), 2015.

[58] Weikang Qian, Xin Li, Marc D Riedel, Kia Bazargan, and David J Lilja. *An architecture for fault-tolerant computation with stochastic logic.* IEEE transactions on computers, 2010.

[59] Moinuddin K Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J Nair, and Onur Mutlu. *AVATAR: A variable-retention-time (VRT) aware refresh for DRAM systems.* Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2015.

[60] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. *Cheetah: Optimizing and accelerating homomorphic encryption for private inference.* International Symposium on High-Performance Computer Architecture (HPCA), 2021.

[61] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. *Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing.* Annual International Symposium on Computer Architecture (ASPLOS), 2017.

[62] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. *DRAMSim2: A cycle accurate memory system simulator.* IEEE computer architecture letters, 2011.

[63] Mehdi Saberi, Reza Lotfi, Khalil Mafinezhad, and Wouter A Serdijn. *Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs.* IEEE Transactions on Circuits and Systems I: Regular Papers, 2011.

[64] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. *ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars.* International Symposium on Computer Architecture (ISCA), 2016.

[65] Hyeonuk Sim and Jongeun Lee. *A new stochastic computing multiplier with application to deep convolutional neural networks.* Design Automation Conference (DAC), 2017.

[66] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition.* International Conference on Learning Representations (ICLR), 2015.

[67] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. *Towards pervasive and user satisfactory cnn across gpu microarchitectures.* International Symposium on High Performance Computer Architecture (HPCA), 2017.

[68] G Edward Suh and Srinivas Devadas. *Physical unclonable functions for device authentication and secret key generation.* Design Automation Conference (DAC), 2007.

[69] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. *Going deeper with convolutions.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

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

[71] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. *Embedding watermarks into deep neural networks.* International Conference on Multimedia Retrieval (ICMR), 2017.

[72] Adrien F Vincent, Jérôme Larroque, Nicolas Locatelli, Nesrine Ben Romdhane, Olivier Bichler, Christian Gamrat, Wei Sheng Zhao, Jacques-Olivier Klein, Sylvie Galdin-Retailleau, and Damien Querlioz. *Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems.* IEEE transactions on biomedical circuits and systems, 2015.

[73] Siyue Wang, Xiao Wang, Pu Zhao, Wujie Wen, David Kaeli, Peter Chin, and Xue Lin. *Defensive dropout for hardening deep neural networks under adversarial attacks.* International Conference on Computer-Aided Design (ICCAD), 2018.

[74] Xingbin Wang, Rui Hou, Boyan Zhao, Fengkai Yuan, Jun Zhang, Dan Meng, and Xuehai Qian. *Dnnguard: An elastic heterogeneous dnn accelerator architecture against adversarial attacks.* International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.

[75] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. *Metal–oxide RRAM.* Proceedings of IEEE, 2012.

[76] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. *Self-training with noisy student improves imagenet classification.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2020.

[77] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. *Aggregated residual transformations for deep neural networks.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[78] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. *Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks.* International Symposium on Computer Architecture (ISCA), 2019.

[79] Chih-Wei Stanley Yeh and S Simon Wong. *Compact one-transistor-N-RRAM array architecture for advanced CMOS technology.* IEEE Journal of Solid-State Circuits, 2015.

[80] Xianwei Zhang, Youtao Zhang, Bruce R Childers, and Jun Yang. *Exploiting DRAM restore time variations in deep sub-micron scaling.* Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015.

[81] Xianwei Zhang, Youtao Zhang, Bruce R Childers, and Jun Yang. *Restore truncation for performance improvement in future DRAM systems.* International Symposium on High Performance Computer Architecture (HPCA), 2016.

[82] Youtao Zhang, Jun Yang, Yongjing Lin, and Lan Gao. *Architectural support for protecting user privacy on trusted processors.* The Workshop on Architectural Support for Security and Anti-Virus, In conjunction with the 11th ASPLOS, 2005.

[83] Lei Zhao, Youtao Zhang, and Jun Yang. *AEP: An error-bearing neural network accelerator for energy efficiency and model protection.* International Conference on Computer-Aided Design (ICCAD), 2017.

[84] Lei Zhao, Youtao Zhang, and Jun Yang. *SCA: a secure CNN accelerator for both training and inference.* Design Automation Conference (DAC), 2020.

[85] Lei Zhao, Youtao Zhang, and Jun Yang. *Flipping Bits to Share Crossbars in ReRAM-Based DNN Accelerator.* International Conference on Computer Design (ICCD), 2021.

[86] Lei Zhao, Youtao Zhang, and Jun Yang. *SRA: A Secure ReRAM-based DNN Accelerator*. Design Automation Conference (DAC), 2022.