FeatureFence: A Regularization Approach for Energy-Efficient Secure Inference on Edge NPUs

Sachintha Kavishan Jayarathne
Department of Electrical and Computer Engineering
University at Albany, SUNY
Albany, NY, USA
kjayarathne@albany.edu

Seetal Potluri
Department of Electrical and Computer Engineering
University at Albany, SUNY
Albany, NY, USA
spotluri@albany.edu

***Abstract*—Feature-snooping attacks (FSA) are very powerful for reverse engineering machine learning models running on neural processing units (NPUs). While memory encryption is an effective countermeasure for cloud devices, the increased data movement causes significant overheads, making it inefficient for edge devices. We make a crucial observation that features dominate the off-chip memory accesses in edge NPUs and propose *FeatureFence*, which eliminates and compensates for feature encryption via a regularization approach to protect against FSA during inference. Our approach creates neuron pairs in the first layer called *couples*, and equates weights and biases of neurons within each couple, thereby making reverse engineering mathematically impossible beyond the first layer. During *FeatureFence* training, the nature of perturbations is gradually learnt across epochs, leading to graceful recovery of functional accuracy. When implemented across a wide range of neural network models mapped to the `Eyeriss` architecture, on average, *FeatureFence* is able to reduce energy overheads by $\approx 41\%$ when compared to *GuardNN*.**

***Index Terms*—Neural Processing Units, Memory Encryption, Energy-Efficiency, Regularization**

I. Introduction

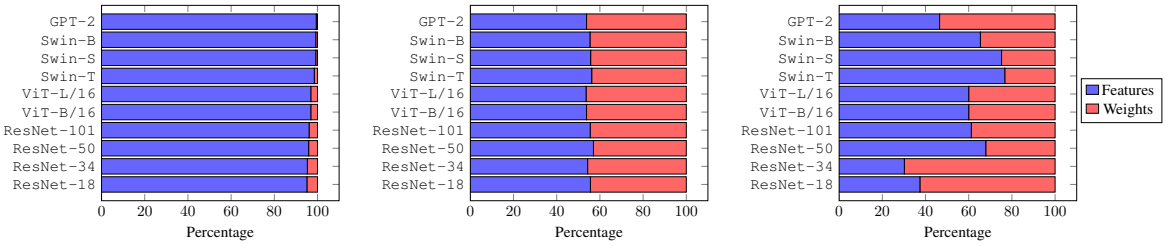
In recent years, most mobile chip design companies have been providing neural processing units (NPUs) to accelerate on-device machine learning (ML) [1]–[5]. Compared to CPUs and GPUs, NPUs are much faster and more energy-efficient for edge devices [6], due to their high density of processing elements (PEs), high resource utilization, and simplified control structure. However, the superior performance comes at the expense of increased privacy risks for the models, which are considered valuable intellectual property (IP) due to the significant R&D investments required to train them.

Unlike software, the internal layers' output features travel back and forth between NPU and the external memory [7], making on-device models vulnerable to *feature snooping attacks* (FSA) [8]. *Memory encryption*, that relies on computational hardness of standard cryptographic primitives is an excellent choice for protection against FSA on cloud devices [9]. However, the energy overhead increases significantly for edge devices, where data movement dominates computation [7]. Table I shows, on average, an order-of-magnitude increase in off-chip memory accesses for an edge device (`Eyeriss`) compared to a cloud device (`TPU`) across a wide range of models and dataflows.

TABLE I
OFF-CHIP MEMORY ACCESSES IN NPUS: WS, RS, IS, AND OS CORRESPOND TO WEIGHT STATIONARY, ROW STATIONARY, INPUT STATIONARY, AND OUTPUT STATIONARY DATAFLOWS, RESPECTIVELY. IN WS/RS, WS IS FOR `TPU` AND RS FOR `Eyeriss`, RESPECTIVELY.

Model	Dataflow	`TPU`	`Eyeriss`	Increase
`ResNet-18`	WS/RS	36,910,320	235,779,928	**6.4**×
	IS	37,491,480	226,828,557	**6.1**×
	OS	33,606,724	79,616,060	**2.4**×
`ResNet-34`	WS/RS	64,612,212	465,935,656	**7.2**×
	IS	67,803,572	451,934,829	**6.7**×
	OS	59,714,296	172,766,620	**2.9**×
`ResNet-50`	WS/RS	72,971,572	618,782,192	**8.5**×
	IS	74,077,032	584,123,864	**7.9**×
	OS	66,700,080	299,391,633	**4.5**×
`ResNet-101`	WS/RS	114,281,300	1,141,579,248	**10**×
	IS	121,079,210	1,093,367,160	**9**×
	OS	106,129,778	580,977,219	**5.5**×
`ViT-B/16`	WS/RS	16,981,112	251,629,166	**14.8**×
	IS	17,084,012	247,784,126	**14.5**×
	OS	12,206,442	190,813,906	**15.6**×
`ViT-L/16`	WS/RS	32,379,872	436,581,306	**13.5**×
	IS	32,494,062	430,105,978	**13.2**×
	OS	23,519,710	330,412,426	**14.1**×
`Swin-T`	WS/RS	74,890,680	714,509,162	**9.5**×
	IS	56,692,860	699,398,474	**12.3**×
	OS	45,326,464	445,249,034	**9.8**×
`Swin-S`	WS/RS	132,988,945	1,379,410,154	**10.4**×
	IS	91,138,266	1,362,565,706	**15**×
	OS	92,293,344	877,076,234	**9.5**×
`Swin-B`	WS/RS	173,517,288	2,441,671,562	**14.1**×
	IS	151,397,360	2,407,308,298	**15.9**×
	OS	121,699,306	1,764,946,474	**14.5**×
`GPT-2`	WS/RS	146,063,433	3,233,955,840	**22.1**×
	IS	201,900,202	3,235,151,872	**16**×
	OS	68,862,000	3,218,940,960	**46.8**×

Since every read/write access necessitates a costly decryption/encryption operation, respectively, the number of off-chip memory accesses correlate directly to the energy overhead. Therefore, it is desirable to evade encryption wherever possible. Weight encryption is inevitable to protect the model's privacy directly. On the other hand, features leak weight-related information through *small-signal analysis* (SSA). In SSA, constraint systems are solved to generate input queries needed to extract beyond the first layer. These constraint systems for layer-i ($i > 1$) depend on the weights and biases of neurons in prior layers, i.e., $1 \ldots (i-1)$. *This presents the possibility to eliminate feature encryption and replace that with weight perturbations during training to make these constraint systems unsolvable.*



(a) Row-Stationary (RS) (b) Input-Stationary (IS) (c) Output-Stationary (OS)

Fig. 1. Profiling off-chip memory accesses for various dataflows in `Eyeriss`.

Weight perturbations were successfully applied in the past to improve adversarial robustness [10], [11] and close the generalization gap [12]–[14] of neural networks. These works exploit regularization to inject noise in either inputs or weights to achieve the specific research objective. To date, no work has exploited regularization to protect model IPs against FSA. Further, the added weight perturbations should not affect the model performance during on-device inference. Finally, features contribute significantly to the off-chip memory accesses, across various dataflows, as shown in Figure 1, making feature encryption a major contributor to the energy overhead. These observations compel us to pose the following research questions (RQs).

RQ1: Is it possible to eliminate feature encryption and compensate with regularization of the weight loss landscape during training to make the constraint systems unsolvable during inference?

RQ2: Is it possible to achieve maximum drop in adversary's reverse engineering accuracy, with minimum drop in the designer's model accuracy?

To address these research questions, we propose *FeatureFence*, the first defense to protect intermediate features from leaking weights during inference. In this context, (a) the **designer** is the one who trains the model, and (b) the **adversary** the one who has access to the device and resources to snoop the memory bus while the device is performing inference. The following are the main contributions of our work:

- For trusted execution on NPUs, we propose replacing feature encryption with a regularization approach during training.
- The designer creates neuron pairs in layer-(1) called *couples*, and equates weights and biases of both neurons within each couple, to ensure the constraint systems fail for layer-(2), and hence naturally for all further layers.
- During *FeatureFence* training, the nature of perturbation is gradually learnt across epochs, leading to security guarantees and recovery of functional accuracy.
- Through empirical evaluation on a wide variety of neural network workloads mapped to the `Eyeriss` architecture, on average, *FeatureFence* demonstrated a $\approx 41\%$ reduction in energy overheads when compared to *GuardNN* [9].

II. Related Work

Existing countermeasures can be categorized into: (a) passive, and (b) active ones. The passive defenses can be sub-categorized into *watermarking* and *fingerprinting* methods. While watermarking aims to embed a signature into the model unique to the model owner, fingerprinting [15], [16] aims at copyright protection by embedding a signature unique to the authorized user. Watermarking can be broadly categorized into white-box [17]–[23] and black-box [24]–[30], in terms of the training function. Some techniques are applicable to both settings [31]–[33]. While mostly, watermarking is invasive, there are non-invasive alternatives as well [31]. While passive defenses are attractive because of their little overheads, they cannot prevent model abuse.

On the contrary, the active defenses are capable of prevention and can be sub-categorized into trusted execution environments (TEEs), and shuffling, dummy insertion, and randomization (SDR) defenses. TEEs provide physical security through *memory encryption* and isolation through *access control*. The TEEs of NPUs can be at *training time* [34], *inference time* [35]–[45], or both [9], [46]–[49]. In this paper, we focus on TEEs that aim at protecting confidentiality during inference through memory encryption [9], [35], [44]–[46], [48], [50], [51]. The SDR defenses [52], [53] and the sNPU architecture [36] aim at confusing the attacker in terms of memory patterns, and protection of in-memory structures, respectively, and hence orthogonal to memory encryption.

While NeuroPlug [51] protects against architectural reverse engineering with memory encryption in place, it does not consider energy-efficiency. Likewise, SparseLock [48] improves performance, but does not address energy-efficiency. While GuardNN [9] demonstrates an energy overhead of $< 3\%$ for a cloud device, the off-chip memory accesses and hence energy overhead are significantly higher in edge devices. Our work addresses this important limitation by retaining weight encryption but eliminating feature encryption and compensating it with a customized regularization approach to improve energy-efficiency of edge devices.

III. System Architecture and Threat Model

The NPU receives instructions from the host CPU and the model parameters from an independent weight memory. Following prior work in NPU memory encryption, we assume the NPU accelerator, the host CPU, and caches are included in the Trusted Computing Base (TCB). In contrast, the off-chip

host memory resides outside the TCB [9]. We assume a strong adversary with full control over the OS or privileged software and the ability to snoop on off-chip buses. For each query, the accelerator is configured to execute the classification task, and the results are snooped on the off-chip bus using a transitioning trigger (memory read/write [54]) signal. Following prior work [55], we assume the structure of the network is already known.

IV. The Feature Snooping Attack (FSA) [8]

Unlike cryptanalytic extraction [55]–[57], where only the output confidence scores are available, the FSA adversary also has access to the intermediate layer outputs. Assume the model f is k-deep and composed of linear transformations f^i, given as $f = f^1 \circ f^2 \circ \dots f^k$. Note f^i includes both the linear transformations as well as any following non-linear transformations, e.g. activation, pooling, etc. Let the corresponding model be composed as $\mathbf{\Theta} = \left[\mathbf{\Theta}^1 | \mathbf{\Theta}^2 \dots | \mathbf{\Theta}^k\right]$. The feature snooping adversary approaches the extraction problem iteratively by observing output of arbitrary intermediate layer-(i), $F^i = f^1 \circ f^2 \dots \circ f^i, 1 \leq i \leq k$.

Key Observation 1: For $i > 1$, output of layer-(i) is directly observable but input to layer-(i) is not directly controllable. Inputs to layer-(i) for $i > 1$, needs to be indirectly controlled by applying suitable inputs to layer-(1) and executing until layer-($i-1$).

Constraint System (CS) Assisted Query Search for Deeper Layers[1]**.** The feature snooping adversary formulates the query search to extract weight-(k) of any neuron in layer-(i) as a solution F^0 to the following constraint system [8]:

$$\texttt{CS(i-1,k)}: (\hat{\mathbf{w}}_{\mathbf{k}}^{\mathbf{i-1}})^\intercal F^{i-2} > -1 \times \hat{b}_k^{i-1} + \eta \quad (1)$$

$$(\hat{\mathbf{w}}_{\mathbf{j}}^{\mathbf{i-1}})^\intercal F^{i-2} < -1 \times \hat{b}_j^{i-1} + \gamma, j \neq k \quad (2)$$

$$F^m = max(0, \hat{\mathbf{W}}^{\mathbf{i-1}} F^{m-1} + \hat{b}_k^{i-1}), 2 \leq m < i-1 \quad (3)$$

where, (a) Equation (1) corresponds to activation of neuron-(k) in layer-($i-1$) using a large positive η, (b) Equation (2) to the deactivation of rest of the neurons in layer-($i-1$) with a negative γ, and (c) Equation (3) corresponds to input-output relationships of all neurons in prior layers, i.e., layers $1 \dots (i-2)$.

A. Sample Attack Illustration for Deeper Layers

We walk through FSA using a simple 2-deep fully connected network (FCN) $\mathbf{\Theta} = \left[\mathbf{\Theta}^1 | \mathbf{\Theta}^2\right]$ trained on the XOR dataset, with 3 and 2 neurons in the first and second layers, respectively. To extract $w_{1,1}^2$, selective neuron activation translates to searching for a query that activates only n_1^1 and n_1^2 and deactivates n_2^1 and n_3^1. This requires solving CS(1,1) as follows:

$$\begin{bmatrix} \hat{w}_{1,1}^1 & \hat{w}_{1,2}^1 \\ -1 * \hat{w}_{2,1}^1 & -1 * \hat{w}_{2,2}^1 \\ -1 * \hat{w}_{3,1}^1 & -1 * \hat{w}_{3,2}^1 \end{bmatrix} [x\ y]^\intercal > \begin{bmatrix} -1 * \hat{b}_1^1 + \eta \\ \hat{b}_2^1 - \gamma \\ \hat{b}_3^1 - \gamma \end{bmatrix}$$

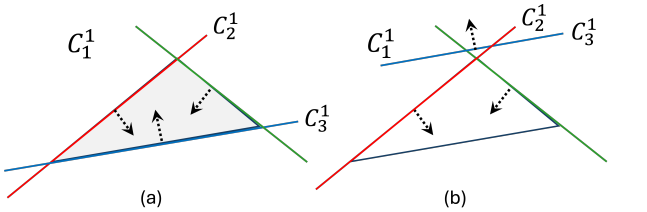


Fig. 2. Nullifying the feasible region for a constraint system by perturbing the bias parameter of neuron n_3^1.

B. Broader Applicability

Although more complex models, such as convolutional neural networks and transformers, use higher-dimensional filters, more sophisticated activations, and other nonlinearities, the concept of FSA remains universal.

V. FeatureFence: A Training-Stage Defense

The satisfiability of the underlying constraint systems is a prerequisite for the success of FSA. We propose *FeatureFence*, a *regularization* approach based on the theme of *neuron coupling* during the training phase, to render the constraint systems *unsatisfiable* during inference.

A. Motivation

Figure 2(a) shows the hyperplanes and feasible region for CS(1,1) discussed above. Let C_1^1, C_2^1, and C_3^1 be the hyperplanes corresponding to n_1^1, n_2^1, and n_3^1, respectively. Assuming the *designer* perturbs the bias parameter value of neuron n_3^1 during training, in such a way as to move the hyperplane C_3^1 to a new position, as shown in Figure 2(b), rendering the feasible region for this new constraint system to $\varnothing$. This makes it impossible to extract $w_{1,1}^2$ and $w_{2,1}^2$.

Let L_n and $L_n^{'}$ be validation losses prior to, and after perturbation, respectively, at the end of epoch-n. During training, the values of $|L_n^{'} - L_n|$ were ≈ 0.00015, 0.00003, and 0.000026, for $n = 1$, 2, and 3, respectively. This indicates that the impact of perturbation gradually reduces and becomes negligible with epoch evolution.

Key Observation 2: Weight encryption should be retained, but feature encryption can be replaced with weight perturbations during training, to defend against FSA during inference, with minimal impact on the model's accuracy.

B. Neuron Coupling

The designer injects weight perturbations into layer-(1) to make it infeasible to solve for layer-(2), effectively rendering it impossible to extract the remaining layers. The motivational example discussed above exploited this principle to protect

[1]In layer-(x), n_y^x corresponds to neuron-(y), $w_{y,z}^x$ to weight-(z) of neuron-(y), b_y^x to the bias value of neuron-(y), $\mathbf{w}_y^x$ to the weight vector of neuron-(y), and $\mathbf{W}^x$ to the layer weight matrix.

one of the weights in layer-(2). We generalize this behavior to multiple weights using the idea of *couples*.

Lemma 1: Given a constraint system `CS(1,k)`, it is unsatisfiable if and only if there are at least two constraints in `CS(1,k)` that conflict each other.

In `CS(1,k)`, we make three observations: (a) each constraint corresponds to a neuron in layer-(1), (b) there is only one ">" constraint and rest are "<" constraints, and (c) the ">" constraint corresponds to neuron n_k^1. Thus, making weights and biases equal for a pair of neurons in layer-1, which we call *couple*, will certainly create a pair of conflicting constraints when the couple includes neuron n_k^1. Hence, according to *Lemma 1*, `CS(1,k)` is unsatisfiable after creating a couple in layer-1 that includes neuron n_k^1.

Lemma 2: `CS(1,k)` is unsatisfiable in the presence of a couple in layer-1 that includes neuron n_k^1.

Considering the `XOR` neural network discussed earlier, without loss of generalization if we pair neurons n_1^1 and n_2^1 to form a couple (n_1^1, n_2^1), then both `CS(1,1)` and `CS(1,2)` transform to the following:

$$\begin{bmatrix} \hat{w}_{1,1}^1 & \hat{w}_{1,2}^1 \\ -1 * \hat{w}_{1,1}^1 & -1 * \hat{w}_{1,2}^1 \\ -1 * \hat{w}_{3,1}^1 & -1 * \hat{w}_{3,2}^1 \end{bmatrix} [x\ y]^\intercal > \begin{bmatrix} -1 * \hat{b}_1^1 + \eta \\ \hat{b}_1^1 - \gamma \\ \hat{b}_3^1 - \gamma \end{bmatrix}$$

Since the first two constraints in this system conflict each other, both `CS(1,1)` and `CS(1,2)` are unsatisfiable, leading to impossibility of extracting for $w_{1,1}^2$, $w_{1,2}^2$, $w_{2,1}^2$, and $w_{2,2}^2$. The original classifier accuracy is 96.7% and the adversary's reverse-engineering accuracy in the presence of couple is 52.3%, which resembled a random XOR classifier that had an accuracy of 49.9%. Thus, *neuron coupling* has significantly thwarted the adversary's reverse-engineering ability.

Theorem 1: Given a couple $(n_{j_1}^1, n_{j_2}^1)$, `CS(1,`j_1`)` and `CS(1,`j_2`)` are both unsatisfiable if the model parameters of both neurons in the couple are made equal.

Proof Sketch: Given a system of linear constraints, each constraint represents a hyperplane that divides the real number space into two half-spaces. The solution to the constraint system lies in the overlapping region of the half-spaces defined by all the constraints. Given that the constants in each constraint correspond to layer-(1) weights and biases, making weights and biases of neurons $n_{j_1}^1$ and $n_{j_2}^1$ equal will create a pair of conflicting constraints and thereby making the overlap equal to $\varnothing$ for both `CS(1,`j_1`)` and `CS(1,`j_2`)`.

Key Observation 3: Since layer-(i)'s extraction relies on layer-($i-1$)'s weights, protecting layer-(2)'s weights naturally protects weights of all the further layers.

C. Regularization Approach to Create Couples

We train the model f_Θ using standard mini-batch optimization with early stopping, as summarized in Algorithm 1. The dataset was divided into training, validation, and test sets $(\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}, \mathcal{D}_{\text{test}})$, for model training, early stopping, and for the final model performance check, respectively. In each epoch t, we iterate over mini-batches (x, y) of size B loaded from $\mathcal{D}_{\text{train}}$. The gradient of the loss ℓ was computed for each mini-batch with respect to the recent parameters Θ, and updated them via the chosen optimizer $\mathcal{O}$. This first phase of Algorithm 1 follows the standard supervised learning loop.

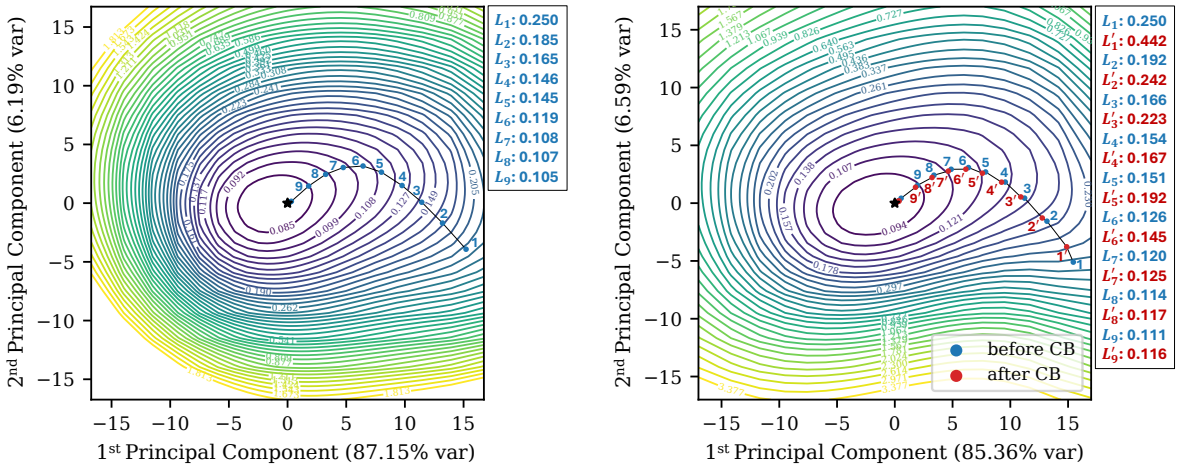
Algorithm 1: *FeatureFence* with early stopping

Input: Model f_Θ; train $\mathcal{D}_{\text{train}}$; validation $\mathcal{D}_{\text{val}}$; test $\mathcal{D}_{\text{test}}$ datasets; loss ℓ; optimizer $\mathcal{O}$; batch size B; max epochs $T_{\max}$; Perturbation factor: α
Result: Best parameters $\Theta^\star$; Performance metric M on $\mathcal{D}_{\text{val}}$ and $\mathcal{D}_{\text{test}}$
Initialize: $\Theta^\star \leftarrow \Theta$; $M_{\text{best}} \leftarrow -\infty$;
for $t \leftarrow 1$ **to** $T_{\max}$ **do**
  `// TRAIN on` $\mathcal{D}_{\text{train}}$
  foreach *mini-batch* (x, y) *with size* B **do**
    Update $\Theta \leftarrow \mathcal{O}\Big(\Theta, \nabla_\Theta\, \ell\big(f_\Theta(x),\, y\big)\Big)$
  `// MODEL PERTURBATION`
  `PerturbWeightsCallback`(f_Θ, α)
  `// VALIDATION on` $\mathcal{D}_{\text{val}}$
  Evaluate $M_t \leftarrow M\big(\mathcal{D}_{\text{val}};\, \Theta\big)$
  `// EARLY STOPPING`
  if $M_t \geq M_{best}$ **then**
    $M_{\text{best}} \leftarrow M_t$; $\Theta^\star \leftarrow \Theta$; *Save_Checkpoint*$(\Theta^\star)$
`// PERFORMANCE of` $f_{\Theta^\star}$ `on` $\mathcal{D}_{\text{val}}$ `and` $\mathcal{D}_{\text{test}}$
$M_{\text{val}}^\star \leftarrow M\big(\mathcal{D}_{\text{val}};\, \Theta^\star\big)$; $M_{\text{test}}^\star \leftarrow M\big(\mathcal{D}_{\text{test}};\, \Theta^\star\big)$

After completing the training for each epoch, Algorithm 1 makes the key modification to f_Θ using the `PerturbWeightsCallback()` function. This perturbs layer-(1)'s trainable parameters with the perturbation factor α, which indicates the number of couples. The value α is designer-configurable, and helps achieve the desired level of security, with minimal impact on the classifier accuracy. Adding weight perturbations is followed by a patience-based early stopping mechanism to prevent overfitting and refine a single model $f_{\Theta^\star}$. This model is well-supported by its performance on the validation dataset $\mathcal{D}_{\text{val}}$, and is influenced by the value of α. Ultimately, we evaluated the performance $M_{\text{best}}^\star$ of the best model $f_{\Theta^\star}$ on the test dataset $\mathcal{D}_{\text{test}}$.

VI. Implementation

This section demonstrates the effectiveness of *FeatureFence* in improving energy-efficiency and reducing the adversary's reverse engineering accuracy, with little impact on the model's functional accuracy. We use the following metrics for validation on various benchmarks: (a) **Designer's accuracy**: the functional accuracy of the actual model in the field, as trained by the *designer*; (b) **Reverse-engineering accuracy**: the accuracy of the extracted model by the *adversary*.



(a) $\alpha = 0$ (b) $\alpha = 8$

Fig. 3. Optimization trajectory on the weight loss landscape.

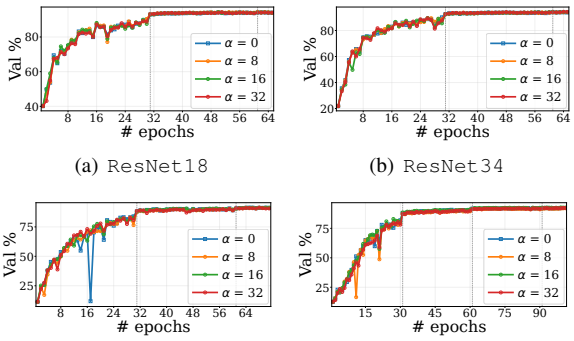
A. Experimental Setup

We used `PyTorch` in a high-performance computing (HPC) based multi-GPU training environment with an AMD EPYC 7513 32-core processor and eight NVIDIA A100 GPUs with 80GB memory per GPU and using the `DataParallel` feature from `torch.nn` library. We implemented model perturbations as a custom `PerturbWeightsCallback()` callback routine, which accesses layer-(1) weights and biases and performs neuron coupling. Leveraging `PyTorch's` flexible API, we seamlessly integrated this callback routine into the training loop, calling it at the end of each epoch.

B. Optimization Paths on Loss Landscape

To elucidate the neural network's response to perturbations, we visualized the weight loss landscape [58] with the optimization path. We overlaid the corresponding optimization trajectory from the first epoch until the best model was found, using early stopping to prevent overfitting and ensure robustness. Owing to the optimization trajectories lying in extremely low-dimensional spaces, we utilized the dominant principal component directions [58] to validate the low dimensionality and to produce compelling visualizations under perturbations. The model was evaluated with a non-augmented transform pipeline and a non-shuffled DataLoader. Training and loss landscapes calculations were run with a fixed random seed to ensure the reproducibility.

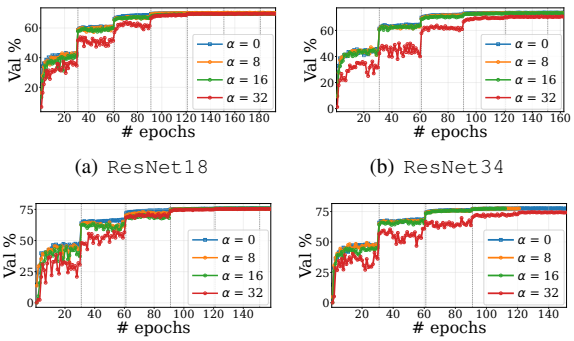
Figure 3 shows this visualization for a simple fully connected network with shape `784-64-32-10` targeting `MNIST`, both in the absence and presence of perturbation. Figure 3(b) suggests that weight perturbation causes significant gap between L'_n and L_n for initial values of n, but the gap gradually closes out. Additionally, a comparison with Figure 3(a) shows that the gap between $\alpha = 8$ and $\alpha = 0$ cases is not significant. This indicates graceful recovery of functional accuracy with *FeatureFence* training.



(a) `ResNet18` (b) `ResNet34`

(c) `ResNet50` (d) `ResNet101`

Fig. 4. *FeatureFence* for CNNs – `CIFAR-10`



(a) `ResNet18` (b) `ResNet34`

(c) `ResNet50` (d) `ResNet101`

Fig. 5. *FeatureFence* for CNNs – `ImageNet-1K`.

C. Implementation on Convolutional Neural Networks (CNNs)

Couple Formation. In feedforward networks, a *couple* involved pairing two unvisited neurons in layer-(1), injecting perturbations into their weights and biases, and marking them as `visited`. Despite the increase in filter dimensions, the formulation can be directly extended by replacing the weighted summation portion of each constraint by a convolution operation. Moreover, the constraint formulation is always performed for one of the features in the neuron's output 2D

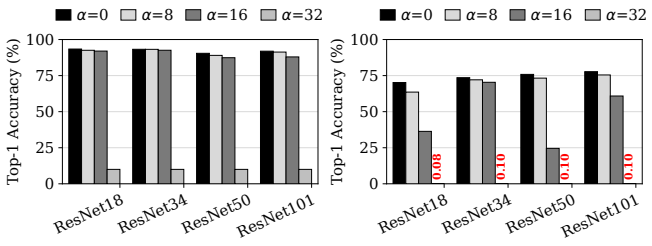
TABLE II
FeatureFence's DESIGNER ACCURACY (TOP-1) FOR CNNS WITH CIFAR-10.

Model	$\alpha = 0$	$\alpha = 8$	$\alpha = 16$	$\alpha = 32$
ResNet-18	93.41	93.40	93.38	93.54
ResNet-34	93.24	93.21	93.42	93.24
ResNet-50	90.45	90.43	90.37	90.09
ResNet-101	91.93	91.62	91.63	91.38

TABLE III
FeatureFence's DESIGNER ACCURACY FOR CNNS WITH IMAGENET-1K.

Model	$\alpha = 0$		$\alpha = 8$		$\alpha = 16$		$\alpha = 32$	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
ResNet-18	70.19	89.36	70.34	89.42	69.57	89.21	69.67	89.30
ResNet-34	73.56	91.53	73.44	91.52	73.32	91.47	70.50	89.77
ResNet-50	75.84	92.75	75.54	92.77	75.86	92.80	75.48	92.59
ResNet-101	77.72	93.83	77.56	93.79	77.45	93.62	74.37	92.11

Note. Matches the original TorchVision's model accuracies [59] .



(a) CIFAR-10 (b) ImageNet-1K

Fig. 6. Reverse-Engineering Accuracy (Top-1) for CNNs

array. Intuitively, the constraint system formulation can be extended to the RGB case by replacing the left hand side in (1), (2), and (3) corresponding to CS(i-1, k) with a three-dimensional convolution. Another important observation is that, unlike a fully connected network, a CNN neuron is a 2D array of features, thereby increasing the source of leakage. Nevertheless, since the formulation is generic, the conflicting constraints indicate that, for all output pixels, the injected perturbations will fail the constraint system.

Results. Figures 4 and 5 show the evolution of validation accuracy after callback, with epochs, in the presence of *FeatureFence* for $\alpha = 0, 8, 16, 32$ cases, for deep CNN models trained using CIFAR-10, and ImageNet-1K datasets, respectively. A general observation is that after the callback, the validation accuracy degrades during the initial epochs for $\alpha \neq 0$. And as epochs progress, the gap systematically reduces and eventually approaches convergence with the $\alpha = 0$ case. This happens because the backpropagation step following each callback, gradually learns about the nature of the injected perturbations. As a result, after the last callback, the security guarantees discussed earlier will be achieved, with minimal impact on the designer's accuracy.

Table II shows the Top-1 final test accuracy achieved during *FeatureFence's* training for the CIFAR-10 dataset, while Table III shows Top-1 and Top-5 for ImageNet-1K. It is worth noting that the impact on the designer's accuracy is minimal. Figure 6 shows the adversary's reverse-engineering accuracy in the presence of *FeatureFence*. The $\alpha = 0$ case corresponds to "no perturbation" scenario and provided for reference. The significant drops for the $\alpha = 32$ case

TABLE IV
FeatureFence's DESIGNER ACCURACY FOR TRANSFORMERS.

Dataset	Model	$\alpha = 0$	$\alpha = \alpha_{max}$	Drop
CIFAR-10	ViT-B/16	98.14	97.94	0.2%
	ViT-L/16	98.06	98.17	−0.11%
	Swin-T	98.02	97.65	0.37%
	Swin-S	98.31	98.12	0.19%
	Swin-B	98.10	98.28	−0.18%
	SwinV2-T	97.71	97.57	0.14%
	SwinV2-S	98.16	98.04	0.12%
	SwinV2-B	98.46	98.31	0.15%
CIFAR-100	ViT-B/16	87.26	87.17	0.09%
	ViT-L/16	87.25	87.13	0.12%
	Swin-T	87.56	87.36	0.2%
	Swin-S	89.20	88.80	0.4%
	Swin-B	89.18	89.26	−0.08%
	SwinV2-T	87.23	86.78	0.45%
	SwinV2-S	88.85	88.93	−0.08%
	SwinV2-B	88.63	88.77	−0.14%
Avg.	-	-	-	**0.12%**

demonstrate the effectiveness of our defense.

D. Extension to Transformer Models

The two main considerations when extending *FeatureFence* to a transformer model is the choice of layer for perturbing weights, and adaptation to the activation function.

Layer Selection. The transformer model is built using a rich collection of positional encoding, attention, feedforward, and normalization layers. Since we have already established strong security proofs for *couples* in the context of feedforward neural networks, we perturb layer-(1) of the first feedforward in the stack, although multi-head attention and normalization layers come ahead in the sequence.

Adapting to `GELU` and `Swish` Activations. The major difference between feedforwards in a transformer, and a regular fully connected network, is the GELU activation:

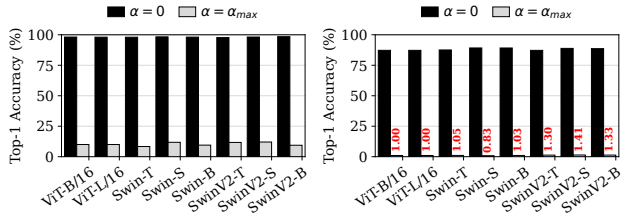
$$\text{GELU}(\text{x}) = x \cdot \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{4}$$

Unlike $\text{ReLU}(\text{x})$, which equals 0, $\forall x \leq 0$, $\text{GELU}(\text{x})$ never reaches 0. However, it is < 0.001 for $x < -4$. Thus, for all practical purposes during SSA, it can be approximated to 0 for $x < -4$. Hence, the modified constraint system for CS(i-1, k) adds a -4 on the right hand side of (1), (2), and (3). Since all hyperplanes shift by the same amount of -4, forming a *couple* will create a pair of conflicting constraints for GELU as well. Likewise, modern LLMs use Swish activation defined as:

$$\text{Swish}(\text{x}) = x \cdot \sigma(\beta x) = \frac{x}{1 + e^{-\beta \cdot x}} \tag{5}$$

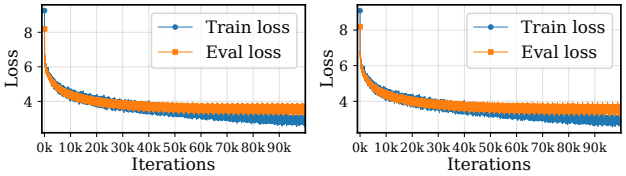
where β is a hyperparameter that controls the function's shape and smoothness. Swish can be approximated to 0, when $x < -9$, thus the constraint system can be modified similar to GELU. Hence, theorem 1 is directly applicable here as well.

Results. Table IV shows the final test accuracy achieved for transformer models during *FeatureFence's* training after early stopping is achieved. Here, α_{MAX} corresponds to the



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

Fig. 7. Reverse-Engineering Accuracy (Top-1) for Transformers.



(a) $\alpha = 0$ (b) $\alpha = \alpha_{MAX}$

Fig. 8. *FeatureFence* Training for GPT-2.

TABLE V
GPT-2 LOSS AND PERPLEXITY.

α	Loss	PPL	RE–Loss	RE–PPL
0	3.73314	41.81	3.73314	41.81
2048	3.74699	42.39	9.27938	10,714.77

maximum number of possible couples in layer-(1) of the first feedforward block in the corresponding transformer model. These strong results demonstrate that our *layer selection strategy* is justified. Similar to CNNs, the impact on the designer's accuracy is also minimal for the transformer models as well. Furthermore, we only considered the $\alpha = \alpha_{MAX}$ case, as the drop in designer's accuracy is negligible even for the extreme case. Figure 7 shows the reverse-engineering accuracy in the presence of *FeatureFence* on various transformer models targeting both CIFAR-10 and CIFAR-100. There are significant drops in reverse-engineering accuracy for $\alpha = \alpha_{MAX}$, with higher drops observed with CIFAR-100, demonstrating the scalability of our defense.

A GPT-2–style transformer with 159.4M parameters was trained on an Expanda-extracted Wikipedia XML dump. Expanda generated the subword vocabulary (vocab.txt) and tokenized both the training and held-out corpora corpus.train.txt (378.9MB) and corpus.test.txt (42.1MB). Training was conducted on a DGX cluster with 8 GPUs using the DistributedDataParallel method for both $\alpha = 0$ and $\alpha = \alpha_{MAX}$. For this GPT-2 model, $\alpha_{MAX} = 2048$, because the feed-forward blocks include 4096 neurons. Figure 8 shows the corresponding loss evolution with iterations for both $\alpha = 0$ and $\alpha = \alpha_{MAX}$ cases, and the negligible impact of weight perturbations on the loss. The loss and perplexity values for $\alpha = 0$ and $\alpha = \alpha_{MAX}$ cases are shown in Table V. While the gap between the two cases is negligible for the designer, it is significant for the adversary.

Energy Savings. The uSystolic-Sim systolic-array simulator [60] was used to compute the energy consumption

TABLE VI
ENERGY SAVINGS OF *FeatureFence* COMPARED TO *GuardNN* [9].

Model	Energy Overhead		Model	Energy Overhead	
	GuardNN [9]	**FeatureFence**		*GuardNN* [9]	**FeatureFence**
ResNet18	53.38%	5.06%	ViT/L-16	25.49%	1.68%
ResNet34	42.95%	4.72%	Swin-T	53.38%	1.48%
ResNet50	34.62%	3.01%	Swin-S	70.77%	0.66%
ResNet101	24.22%	2.8%	Swin-B	46.66%	0.43%
ViT/B-16	35.19%	2.33%	GPT-2	46.14%	0.62%

TABLE VII
IMPACT ON EPOCH COUNTS FOR CNNS.

Dataset	Model	Epochs				% Increase		
		$\alpha = 0$	$\alpha = 8$	$\alpha = 16$	$\alpha = 32$	$\alpha = 8$	$\alpha = 16$	$\alpha = 32$
CIFAR10	ResNet18	42	34	37	42	−19%	−12%	0%
	ResNet34	41	45	45	45	10%	10%	10%
	ResNet50	64	64	66	71	0%	3%	11%
	ResNet101	170	109	101	109	−36%	−41%	−36%
ImageNet	ResNet18	158	164	176	173	4%	11%	10%
	ResNet34	136	127	127	205	−7%	−7%	51%
	ResNet50	146	169	151	175	16%	3%	20%
	ResNet101	150	122	112	186	−19%	−25%	24%
Avg.	-	-	-	-	-	**−6.4%**	**−7.3%**	**11.3%**

TABLE VIII
IMPACT ON EPOCH COUNTS FOR TRANSFORMERS.

Model	CIFAR-10			CIFAR-100		
	$\alpha = 0$	$\alpha = \alpha_{MAX}$		$\alpha = 0$	$\alpha = \alpha_{MAX}$	
	# Epochs	# Epochs	% Inc.	# Epochs	# Epochs	% Inc.
ViT-B/16	23	21	−9%	25	25	0%
ViT-L/16	33	40	21%	17	17	0%
Swin-T	58	38	−34%	63	54	−14%
Swin-S	34	28	−18%	38	38	0%
Swin-B	26	55	112%	45	47	4%
SwinV2-T	70	42	−40%	58	55	−5%
SwinV2-S	29	26	−10%	43	47	9%
SwinV2-B	43	39	−9%	35	45	29%
Avg.			**1.6%**			**2.9%**

of on-device inference for a range of AI models when mapped to the Eyeriss architecture at 32nm node. We instantiate an energy-efficient AES engine at the same technology node from [61], which consumes 22 pJ/bit. The total energy consumed for feature and weight encryptions are then scaled using the off-chip memory accesses obtained from uSystolic-Sim [60] and Eyeriss's bitwidth. Table VI shows the on-chip energy overhead comparisons for all the considered CNN and transformer models. On average, our proposed *FeatureFence* defense saves $\approx$ 41% energy overheads when compared to *GuardNN* [9].

VII. DISCUSSION

Impact on Training Times. Likewise, the impact of *FeatureFence* on epoch counts for various CNN and transformer models is shown in Tables VII and VIII, respectively. We noticed an average impact of $< 3\%$ across all the models. Coming to GPT-2, $\alpha = 0$ and $\alpha = \alpha_{MAX}$ cases completed at 68499 and 61499 iterations, respectively. This effectively means the training actually completed faster in the presence of weight perturbation.

Applicability to Cryptanalytic Extraction. Cryptanalytic extraction [55] looks at the input-output relationship, and in turn relies on intermediate layers to generate critical points. *FeatureFence* ensures that intermediate layers do not leak information, thereby preventing the output from receiving

useful information from them, making our defense naturally resistant to cryptanalytic extraction.

Applicability to Physical Side-Channels. Similar to FSA, for $i > 1$, physical side-channels also rely on constraint satisfaction in layer-$(i-1)$ to extract weights in layer-(i) [62], making *FeatureFence* directly useful.

Impact of Quantization. Recent work demonstrates that requantization-induced errors with quantized models can themselves suppress FSA, further supporting the elimination of feature encryption while retaining weight encryption [63].

VIII. Conclusion

This paper examines an energy-efficient alternative to feature encryption to protect cost-sensitive AI models executing on NPUs from feature-snooping attacks. We developed *FeatureFence*, a regularization approach for training, that exploits *neuron coupling* to protect the features from leaking weights during inference. We demonstrated security guarantees for *couples* in the presence of `ReLU` activation function and adapted to `GELU` and `Swish` as well. Validation across a wide range of models and datasets, demonstrated significant energy savings compared to *GuardNN*, low adversary's reverse-engineering accuracy, and with little impact on the designer's accuracy and training times. This, along with seamless integration with `PyTorch`, makes it attractive for real-world adoption.

References

[1] ARM ML Processor. https://www.arm.com/solutions/artificial-intelligence.

[2] Samsung Neural Processing Unit. https://www.samsung.com/global/galaxy/what-is/npu/.

[3] Qualcomm AI Engine. https://www.qualcomm.com/processors/ai-engine.

[4] Huawei Ascend NPU. https://support.huaweicloud.com/intl/en-us/usermanual-cce/cce_10_0239.html.

[5] Apple Neural Engine. https://www.apple.com/newsroom/2025/10/apple-unleashes-m5-the-next-big-leap-in-ai-performance-for-apple-silicon/.

[6] Tianxiang Tan et al. Deep Learning on Mobile Devices Through Neural Processing Units and Edge Computing. In *IEEE INFOCOM*, pages 1209–1218, 2022.

[7] Yu-Hsin Chen et al. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE JSSC*, 52(1):127–138, 2017.

[8] Seetal Potluri et al. Stealing Neural Network Models through the Scan Chain: A New Threat for ML Hardware. In *IEEE/ACM ICCAD*, pages 1–8, 2021.

[9] Weizhe Hua et al. GuardNN: Secure Accelerator Architecture for Privacy-Preserving Deep Learning. In *ACM/IEEE DAC*, pages 349–354, 2022.

[10] Dongxian Wu, Shu-Tao Xia, and Yisen Wang. Adversarial weight perturbation helps robust generalization. *NeurIPS*, 33:2958–2969, 2020.

[11] Jianhan Xu et al. Weight perturbation as defense against adversarial word substitutions. In *EMNLP*, pages 7054–7063, 2022.

[12] Chris M Bishop. Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1):108–116, 1995.

[13] Li Wan et al. Regularization of Neural Networks using DropConnect. In *ICML*, pages 1058–1066. PMLR, 2013.

[14] Tao Li et al. Efficient generalization improvement guided by random weight perturbation. *arXiv preprint arXiv:2211.11489*, 2022.

[15] Kang Yang et al. MetaFinger: Fingerprinting the Deep Neural Networks with Meta-training. In *IJCAI*, pages 776–782, 2022.

[16] Huili Chen et al. DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models. In *ACM ICMR*, pages 105–113, 2019.

[17] Yusuke Uchida et al. Embedding Watermarks into Deep Neural Networks. In *ACM ICMR*, pages 269–277, 2017.

[18] Jialong Zhang et al. Protecting Intellectual Property of Deep Neural Networks with Watermarking. In *ACM AsiaCCS*, pages 159–172, 2018.

[19] Jiangfeng Wang et al. Watermarking in Deep Neural Networks via Error Back-propagation. *Electronic Imaging*, 32(4):22–1–22–9, 2020.

[20] Xiaoyu Cao et al. IPGuard: Protecting Intellectual Property of Deep Neural Networks via Fingerprinting the Classification Boundary. In *ACM AsiaCCS*, pages 14–25, 2021.

[21] Tianhao Wang et al. RIGA: Covert and Robust White-Box Watermarking of Deep Neural Networks. In *WWW*, pages 993–1004, 2021.

[22] Huili Chen et al. SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems. In *Interspeech*, pages 2312–2316, 2020.

[23] Torsten Krauß et al. ClearStamp: A Human-Visible and Robust Model-Ownership Proof based on Transposed Model Training. In *USENIX Security*, pages 5269–5286, 2024.

[24] Jia Guo and Miodrag Potkonjak. Watermarking deep neural networks for embedded systems. In *IEEE/ACM ICCAD*, pages 1–8, 2018.

[25] Yossi Adi et al. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In *USENIX Security*, pages 1615–1631, 2018.

[26] Jie Zhang et al. Model Watermarking for Image Processing Networks. In *AAAI*, pages 12805–12812, 2020.

[27] Erwan Le Merrer et al. Adversarial Frontier Stitching for Remote Neural Network Watermarking. *Neural Computing and Applications*, 32(13):9233–9244, 2020.

[28] Tianxiang Shen et al. SOTER: Guarding Black-box Inference for General Neural Networks at the Edge. In *USENIX ATC*, pages 723–738, 2022.

[29] Chenqi Xie et al. Deepmark: Embedding watermarks into deep neural network using pruning. In *IEEE ICTAI*, pages 169–175, 2021.

[30] Abhishek Chakraborty et al. DynaMarks: Defending Against Deep Learning Model Extraction Using Dynamic Watermarking. *CoRR*, abs/2207.13321, 2022.

[31] Jialuo Chen et al. Copy, Right? A Testing Framework for Copyright Protection of Deep Learning Models. In *IEEE S&P*, pages 824–841, 2022.

[32] Bita Darvish Rouhani et al. DeepSigns: An End-to-End Watermarking Framework for Ownership Protection of Deep Neural Networks. In *ACM ASPLOS*, pages 485–497, 2019.

[33] Zheng Li et al. How to Prove Your Model Belongs to You: A Blind-Watermark Based Framework to Protect Intellectual Property of DNN. In *IEEE ACSAC*, pages 126–137, 2019.

[34] Husheng Han et al. TensorTEE: Unifying Heterogeneous TEE Granularity for Efficient Secure Collaborative Tensor Computing. In *ACM ASPLOS*, pages 282–297, 2024.

[35] Xingbin Wang et al. NPUFort: A Secure Architecture of DNN Accelerator Against Model Inversion Attack. In *ACM CF*, pages 190–196, 2019.

[36] Erhu Feng et al. sNPU: Trusted Execution Environments on Integrated NPUs. In *ACM/IEEE ISCA*, pages 708–723, 2024.

[37] Manaar Alam et al. NN-Lock: A Lightweight Authorization to Prevent IP Threats of Deep Learning Models. *ACM JETC*, 18(3), 2022.

[38] Nivedita Shrivastava and Smruti R. Sarangi. Securator: A Fast and Secure Neural Processing Unit. In *IEEE HPCA*, pages 1127–1139, 2023.

[39] Sunho Lee et al. TNPU: Supporting Trusted Execution with Tree-less Integrity Protection for Neural Processing Unit. In *IEEE HPCA*, pages 229–243, 2022.

[40] Xunjie Wang et al. TZ-LLM: Protecting On-Device Large Language Models with Arm TrustZone. *arXiv preprint arXiv:2511.13717*, 2025.

[41] Xuanyao Peng et al. SecNPU: Securing LLM Inference on NPU. In *IEEE ICCD*, pages 25–32, 2025.

[42] Myungsuk Moon et al. ASGARD: Protecting On-Device Deep Neural Networks with Virtualization-Based Trusted Execution Environments. In *NDSS*, 2025.

[43] Sarbartha Banerjee et al. Triton: Software-Defined Threat Model for Secure Multi-Tenant ML Inference Accelerators. In *HASP*, page 19–28, 2023.

[44] Aritra Dhar et al. Ascend-CC: Confidential Computing on Heterogeneous NPU for Emerging Generative AI Workloads. *arXiv:2407.11888*, 2024.

[45] Qiushi Li et al. ENIGMA: Low-Latency and Privacy-Preserving Edge Inference on Heterogeneous Neural Network Accelerators. In *IEEE ICDCS*, pages 458–469, 2022.

[46] Weizhe Hua et al. MGX: Near-Zero Overhead Memory Protection for Data-Intensive Accelerators. In *ACM/IEEE ISCA*, pages 726–741, 2022.

[47] Jianyu Jiang et al. CRONUS: Fault-isolated, Secure and High-performance Heterogeneous Computing for Trusted Execution Environment. In *IEEE/ACM MICRO*, pages 124–143, 2022.

[48] Nivedita Shrivastava et al. SparseLock: Securing Neural Network Models in Deep Learning Accelerators. *CoRR*, abs/2311.02628, 2023.

[49] Aritra Dhar et al. GuardAIn: Protecting Emerging Generative AI Workloads on Heterogeneous NPU. In *IEEE S&P*, pages 4155–4172, 2025.

[50] Wei Ren et al. AccShield: A New Trusted Execution Environment with Machine-Learning Accelerators. In *ACM/IEEE DAC*, pages 1–6, 2023.

[51] Nivedita Shrivastava et al. NeuroPlug: Plugging Side-Channel Leaks in NPUs using Space Filling Curves. *arXiv preprint arXiv:2407.13383*, 2024.

[52] Karthik Ganesan et al. BlackJack: Secure machine learning on IoT devices through hardware-based shuffling. *arXiv preprint arXiv:2310.17804*, 2023.

[53] Yuntao Liu et al. Mitigating reverse engineering attacks on deep neural networks. In *IEEE ISVLSI*, pages 657–662, 2019.

[54] Weizhe Hua et al. Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks. In *ACM/IEEE DAC*, pages 4:1–4:6, 2018.

[55] Nicholas Carlini et al. Cryptanalytic Extraction of Neural Network Models. In *CRYPTO*, pages 189–218, 2020.

[56] David Rolnick and Konrad P. Körding. Identifying Weights and Architectures of Unknown ReLU Networks. *CoRR*, abs/1910.00744, 2019.

[57] Matthew Jagielski et al. High Accuracy and High Fidelity Extraction of Neural Networks. In *USENIX Security*, pages 1345–1362, 2020.

[58] Hao Li et al. Visualizing the Loss Landscape of Neural Nets. In *NeurIPS*, volume 31, pages 6391–6401, 2018.

[59] TorchVision Developers. Models and Pre-trained Weights. https://docs.pytorch.org/vision/stable/models.html. ImageNet-1k single-crop Acc@1/Acc@5. Accessed 2025-10-18.

[60] Di Wu and Joshua San Miguel. uSystolic: Byte-Crawling Unary Systolic Array. In *IEEE HPCA*, pages 12–24, 2022.

[61] Yuhao Wang, Leibin Ni, Chip-Hong Chang, and Hao Yu. DW-AES: A domain-wall nanowire-based AES for high throughput and energy-efficient data encryption in non-volatile memory. *IEEE TIFS*, 11(11):2426–2440, 2016.

[62] Gongye Cheng et al. Side-Channel-Assisted Reverse-Engineering of Encrypted DNN Hardware Accelerator IP and Attack Surface Exploration. In *IEEE S&P*, pages 4678–4695, 2024.

[63] Sachintha K. Jayarathne et al. Is Encryption Necessary for Quantized Inference on Neural Processing Units? In *IEEE MWSCAS*, pages 1011–1015, 2025.