MicroPower: Micro Neural Networks for Side-Channel Attacks

Logan Reichling[†], Ryan Evans[†], Mabon Ninan[†], Phuc Mai[†], Boyang Wang[†], Yunsi Fei[§], John M. Emmert[†]
[†]University of Cincinnati, [§]Northeastern University
{*reichlln, evans2ra, ninanmm, maipd*}*@mail.uc.edu, boyang.wang@uc.edu, y.fei@northeastern.edu, john.emmert@uc.edu*

***Abstract*—Side-channel attacks recover encryption keys from a target by analyzing power consumption. Deep learning side-channel attacks require less pre-processing and can defeat countermeasures compared to traditional methods. However, neural networks utilized in side-channel attacks are often complex, which requires substantial storage and memory. In this paper, we propose an iterative pruning algorithm, named MicroPower, which can significantly reduce the size and memory usage of neural networks for side-channel attacks. Our algorithm automatically adjusts pruning parameters of the next iteration based on the results of its previous iteration to derive extremely small neural networks that can still recover keys successfully. We conduct comprehensive evaluation over three existing datasets and demonstrate the effectiveness of our algorithm. For instance, our algorithm can remove 99.86% of parameters in a CNN and the pruned CNN can still recover keys over ASCADv1 dataset with only 321 traces. Given ASCADv2 dataset, we can remove 91.74% of parameters in a ResNet and reveal encryption keys with only 519 traces. Our pruning is also compatible with quantization, which can further reduce the size of a neural network. Moreover, we demonstrate that our pruned neural networks can run efficiently and effectively on embedded devices, including Nvidia Jetsons and AMD/Xilinx ZCU104 FPGA boards.**

I. Introduction

Side-Channel Attacks (SCAs) [1], [2], [3], [4] recover encryption keys from a target (e.g., a microcontroller) based on correlations between power consumptions and intermediate results of encryption, such as AES (Advanced Encryption Standard). Recent studies [5], [6], [7], [8], [9], [10] have shown that Deep Learning Side-Channel Attacks surpass traditional methods in effectiveness, as they can overcome countermeasures, including masking and random delays, and can operate on raw power/EM traces with minimal or no preprocessing.

Despite the promising findings, neural networks adopted in side-channel attacks are often complex with millions of parameters. For instance, the ResNet (Residual Neural Network) utilized over the recent ASCADv2 dataset consists of over 137 million parameters [11]. Executing side-channel attacks with a complex neural network demands substantial storage and memory [12].

In this paper, we introduce *MicroPower*, an *iterative pruning algorithm* that can significantly reduce the size and memory usage of a neural network for side-channel attacks. Specifically, our algorithm removes less important filters in a neural network through multiple iterations, where a neural network is pruned and fine-tuned in each iteration. The pruned neural network from one iteration serves as input for the next iteration. Our algorithm automatically adjusts the pruning parameters for the next iteration based on the results of the previous iteration. Our pruning completes when pruning parameters cannot be further updated or a neural network cannot be further reduced. Our main findings are summarized below:

- We perform comprehensive evaluations of our iterative pruning on three existing datasets, including the ASCADv1 dataset [7], ASCADv2 dataset [11], and TinyPower dataset [12]. Over 3 million power/EM traces are examined. These traces were acquired from microcontrollers (AVR XMEGA and ARM STM32F) and FPGAs (AMD/Xilinx Artix-7) running unmasked or masked AES-128. We leverage two neural networks, including a CNN (Convolutional Neural Network) and a ResNet, from existing studies as two baseline models.
- Experimental results show that our iterative pruning can dramatically reduce the number of parameters in a neural network with a minor impact on attack results. For instance, given the ASCADv1 dataset, our iterative pruning can remove 99.86% of the parameters (from 43 million to 62 thousand) in a CNN and the pruned CNN still recovers one key byte with only 321 traces during the attack. Given the ASCADv2 dataset, we can remove 91.74% of the parameters (from 137 million to 11 million) in a ResNet and reveal all the 16 key bytes with only 519 traces.
- Compared to an existing study [13], which derives the smallest neural networks for Side-Channel Attacks using neural network architecture search, our iterative pruning derives a similar number of parameters in a neural network. On the other hand, our method requires a much shorter search time (3.7$\sim$5.5X faster) and reveals keys with a significantly lower number of traces. Moreover, the pruned neural networks derived by our method can still recover keys in cross-device scenarios while the ones obtained by neural network architecture search cannot.
- We also demonstrate that our iterative pruning is compatible with quantization (more specifically, post-training quantization), which can further reduce the size of a neural network by lowering weights from 32 bits to 8 bits. Moreover, we show the effectiveness of our pruned neural networks on embedded devices, including a Nvidia Jetson Orin Nano and an AMD/Xilinx ZCU104 FPGA board.

Reproducibility. The source code of our design can be found

at https://github.com/UCdasec/MicroPower

II. Background on Side-Channel Attacks

Side-Channel Attacks are categorized into two types: *Profiling Side-Channel Attacks* [3], [4], [5] and *Non-Profiling Side-Channel Attacks* [2], [14]. We focus on profiling attacks in this paper as most of the existing Deep Learning Side-Channel Attacks are profiling.

System Model. As shown in Fig. 1, a Profiling Side-Channel Attack involves two devices: a *training device* and a *test device*. The attacker has complete control over the training device, allowing the creation of a profile/classifier. Specifically, the attacker controls the plaintexts and key on the training device and can capture power or electromagnetic (EM) traces. Conversely, the attacker does not know the key on the test device, but can observe plaintexts and can passively capture its power consumption or EM radiation while these plaintexts are encrypted. The objective of the attacker is to reveal the *unknown key* on the test device.

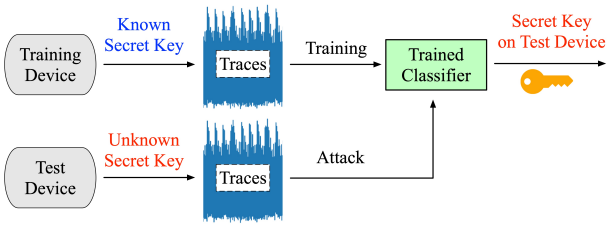


Fig. 1: System model of profiling side-channel attacks.

A Profiling Side-Channel Attack comprises of two phases: the *profiling phase* and the *attack phase*. In the profiling phase, the attacker trains a profile/classifier using labeled traces collected from the training device. These labels correspond to intermediate encryption results, which are derived from the known key and plaintexts following the encryption algorithm. In the attack phase, the attacker obtains unlabeled traces from the test device and uses the trained profile to recover the unknown key. A classifier is a neural network in this study. *In essence, the attacker relies on the trained classifier to predict intermediate encryption results based on power consumption, and then distinguishes the correct key based on confidence scores of intermediate results and given plaintexts.*

Notations. A trace is a sequence of samples collected from a device when it operates one execution of encryption given a key and a plaintext. Each sample is the power measurement or EM radiation at a given timestamp. We use $\mathcal{M}$ and $\mathcal{K}$ to denote the plaintext space and the key space, respectively. Given a plaintext $m \in \mathcal{M}$ and a key $k \in \mathcal{K}$, a device runs an encryption algorithm and a trace $t = (t[1], ..., t[l])$ is recorded, where $t[i]$ is the sample at timestamp i. We leverage z, which is the output of function $\varphi(m, k)$, to denote an intermediate value of encryption, where function $\varphi(\cdot)$ is a leakage step carrying side-channel leakage. *Points of Interest (POIs)* are samples in a trace that are associated with the leakage step.

AES. We focus on attacks targeting AES-128, where the entire encryption key consists of 16 bytes. A side-channel attack on AES-128 typically recovers one key byte each time. If one key byte can be recovered, then it is trivial to recover the remaining bytes by repeating the training and attacks. Hence, we consider key k, plaintext m, or intermediate value z has only one byte. Let $k_1^*, k_2^*, ..., k_{256}^*$ be all the possible 256 key candidates for key k. We consider the `SubBytes` operation of the first round of AES-128 as the leakage step $\varphi(\cdot)$ [12].

Leakage Model. We model side-channel leakage using the *Identity (ID) model*. The ID model assumes that there is a correlation between power consumption and intermediate value z (i.e., the output of `SubBytes` of the first round). As there are 256 possible values for intermediate value z, each trace can have one of these 256 values serving as its label.

Evaluation Metrics. During the attack, the attacker obtains test traces and their associated plaintexts. Given a test trace t and its plaintext m, the trained classifier first outputs a confidence score for every possible label (i.e. every possible intermediate value) by using trace t as the input to the classifier. Then, each score is further assigned to a key candidate following the inverse of `SubBytes` and `AddKey` given plaintext m and each possible intermediate value. The score of every key candidates is further aggregated across test traces. The 256 aggregated scores, one for each key candidate, are then sorted in a descending order.

We leverage *Key Rank* (*or guessing entropy*) [15], [10] and *Measurements To Disclosure* (MTD) to measure the effectiveness of Side-Channel Attacks. Key rank r, where $r \in [1, 256]$, is the rank of the correct key given the sorted scores of all 256 possible key candidates. A key rank of 1 suggests that the attack distinguishes the correct key from other incorrect candidates. MTD indicates the number of test traces needed for the key rank to converge to 1. An attack is more effective if MTD is lower.

III. Our Proposed Design

A. Background in Pruning

Pruning Categories. Neural network pruning, or pruning in short, refers to the process of selectively removing *less important* parameters/weights in a neural network to reduce size and memory usage in prediction [16], [17]. We focus on *structured pruning*, which removes parameters in groups. Specifically, we remove filters at each layer in a neural network. We focus on iterative pruning, which prunes a neural network through multiple iterations rather than pruning a neural network with a single iteration. In other words, the pruned neural network from a previous iteration is utilized as the input for the next iteration to further reduce the number of parameters. Essentially, *our design is an iterative structured pruning.*

Pruning Rate. A pruning rate p indicates the ratio of filters that will be removed to the total number of filters in a given layer. While the pruning rate can be different across layers within an iteration, we keep it the same for all the layers within an iteration for ease of design. Given a pruning rate p and a set of n filters $\mathbf{F} = \{F_1, ..., F_n\}$ at a layer in a neural network, the pruning, in essence, is an optimization algorithm to find

a subset $\mathbf{F}^*$of $\mathbf{F}$ with $\lceil(1-p)\times n\rceil$ filters such that these $\lceil(1-p)\times n\rceil$ filters achieve the maximum sum of importance, which can be formulated as below

$$\arg\max_{\mathbf{F}^*\subset\mathbf{F}} = \sum_{F_j\in\mathbf{F}^*} \alpha(F_j) \tag{1}$$

where $j \in [1,n]$ and $\alpha(\cdot)$ is a pruning score algorithm measuring the importance of each filter.

Filter Score Algorithm. We use an existing filter score algorithm l_2-norm [18], which measures the importance of a filter independently. Given a filter $F_j \in \mathbb{R}^{n'\times s\times s}$, where n' is the number of filters from the previous layer (or the number of input channels if it is the first layer) and s is the kernel size, the l_2 norm of this filter is calculated as

$$\alpha(F_j) = ||F_j||_2 = \sqrt{\sum_{x=1}^{n'}\sum_{y=1}^{s}\sum_{z=1}^{s}(w_j[x][y][z])^2} \tag{2}$$

where $w_j[x][y][z]$ is an element/weight of filter F_j, for $1 \le x \le n'$, $1 \le y \le s$, and $1 \le z \le s$. A higher l_2-norm indicates that a filter is more important. It is worth mentioning that our design also allows other score algorithms to be utilized.

B. The Challenge and Our Main Idea

To design an iterative pruning algorithm, we need to address two key questions. First, *given one iteration, what is the criteria for an algorithm to proceed to the next iteration?* Second, *how many iterations should we perform before we end pruning?* A straightforward solution would be *manually* defining a pruning rate for each iteration and choosing a number of iterations in advance given a dataset. Unfortunately, this manual process would be extremely time-consuming and not scalable given a large search space (i.e., all the possible combinations of the pruning rate and the number of iterations).

We address these two questions above by ***designing an automatic iterative pruning algorithm that is customized for side-channel attacks***. Specifically, given a dataset, a trained complex neural network, and some pruning parameters as inputs, our iterative pruning eventually outputs a pruned neural network after a certain number of iterations. Within each iteration, our algorithm includes two steps: *pruning* and *fine-tuning*. Pruning removes less important filters based on l_2 norms and fine-tuning updates the weights of the pruned neural network of this iteration. For ease of description, we denote the neural network before the pruning of an iteration as the *parent network* and the pruned neural network after the fine-tuning of an iteration as the *child network*. At the end of an iteration, if the accuracy of this child network over validation traces is higher than a threshold, our algorithm marks this iteration as successful and moves on to the next iteration by using this child network as the parent network of the next iteration without updating pruning parameters. Otherwise, our algorithm marks this iteration as unsuccessful and moves on to the next iteration by using the same parent network as input, but adjusting pruning parameters. This threshold, denoted as the *accuracy threshold*, is a parameter that our algorithm defines in advance. It should be higher than random guess (i.e., 1/256 given the ID model) but lower than the accuracy of a well-trained complex neural network. An in-depth discussion regarding the selection of accuracy threshold is presented in Sec. IV.

When pruning parameters need to be updated due to an unsuccessful iteration, our algorithm increases the number of epochs used in fine-tuning (e.g. by 25 epochs; this number is chosen in regards to the original model's hyperparameters) as long as it does not exceed a maximal number of epochs (pre-defined in advance, e.g. 600 epochs) while keeping the current pruning rate. With the updates, our algorithm moves on to the next iteration. However, once the maximal number of epochs is reached given the current pruning rate, our algorithm decreases the pruning rate (e.g. by 0.1) and resets the number of epochs to an initial value (e.g. 150 epochs) for the next iteration.

Algorithm 1: Our Iterative Pruning: MicroPower

Input: D, N, p_{init}, p_{dec}, e_{init}, e_{max}, e_{inc}, acc_{thres}
Output: N_{pruned}
$N_{parent} \leftarrow N$, $N_{child} \leftarrow none$, $e \leftarrow e_{init}$, $p \leftarrow p_{init}$
$flag \leftarrow$ `true`
while $flag ==$ `true` **do**
  if $|N_{parent}| == |N_{child}|$ **then**
    $flag \leftarrow$ `false`, **break**
  end
  $N_{child} \leftarrow$ `Prune&FineTune`(D, N_{parent}, p, e)
  $acc \leftarrow$ `GetAccuracy`(N_{child})
  if $acc \ge acc_{thres}$ **then**
    $N_{parent} \leftarrow N_{child}$
  else
    $e \leftarrow e + e_{inc}$
    if $e \ge e_{max}$ **then**
      $p \leftarrow p - p_{dec}$, $e \leftarrow e_{init}$
      if $p \le 0$ **then**
        $flag \leftarrow$ `false`, **break**
      end
    end
  end
end
$N_{pruned} \leftarrow$ `FineTune`(D, N_{child})

We present the pseudo-code of our algorithm in Algo. 1 based on the above description. We denote the dataset as D, the trained complex neural network as N, the pruning rate as p, the initial pruning rate as p_{init}, the number of epochs as e, the number of maximal epochs as e_{max}, the initial number of epochs as e_{init}, the increment number of epochs as e_{inc}, the pruning rate decrement as p_{dec}, the accuracy threshold as acc_{thres}, and the output, i.e. the pruned neural network, as N_{pruned}. We utilize $|N_{parent}| == |N_{child}|$ to indicate the parent and children neural networks having the same number of parameters. We use `Prune&FineTune`$(\cdot)$ to describe the pruning and fine-tuning process of a single iteration and `GetAccuracy`$(\cdot)$ to present an algorithm to obtain accuracy of a neural network over validation traces,

TABLE I: Overview of Datasets Utilized in Our Evaluation

Target	Dataset Name	No. of Traces	POIs	Channel	Same Device	Cross Device	Countermeasures
ARM STM32F	S1_K1_200k	200k	[1200,2200]	Power	✓		No
	S2_K2_100k	100k	[1200,2200]	Power		✓	No
	S1_K1_150k_EM	150k	[1200,2200]	EM	✓		No
	S2_K2_150k_EM	150k	[1200,2200]	EM		✓	No
	S1_K1_200k_RD	200k	[1200,2200]	Power	✓		Random Delay
	S2_K2_100k_RD	100k	[1200,2200]	Power		✓	Random Delay
AVR XMEGA	X1_K1_200k	200k	[1800,2800]	Power	✓		No
	X2_K2_100k	100k	[1800,2800]	Power		✓	No
	X1_K1_150k_EM	150k	[1800,2800]	EM	✓		No
	X2_K2_150k_EM	150k	[1800,2800]	EM		✓	No
	X1_K1_200k_RD	200k	[1800,2800]	Power	✓		Random Delay
	X2_K2_100k_RD	100k	[1800,2800]	Power		✓	Random Delay
Xilinx Artix-7	F1_K1_200k	200k	[0,100]	Power	✓		No
	F2_K2_200k	200k	[0,100]	Power		✓	No
AVR ATMEGA	ASCADv1	60k	[45400, 46100]	Power	✓	✗	Masking (1st order)
ARM STM32F	ASCADv2	770k	[0,15000]	Power	✓	✗	Masking (3rd order) and Shuffling

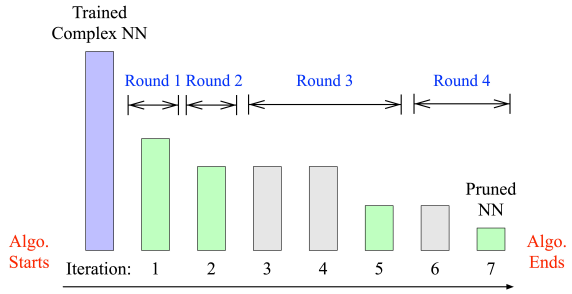


Fig. 2: An example of pruning a complex neural network to a pruned neural network with our algorithm throughout seven iterations. Iteration 1, 2, 5, and 7 are successful and iteration 3, 4, and 6 are marked as unsuccessful.

which do not overlap with training traces. A final fine-tuning (or tempering) step $\texttt{FineTune}(\cdot)$ is applied to further boost the accuracy of the final pruned neural network. An example of how a complex neural network could be pruned through multiple iterations with our algorithm is illustrated in Fig. 2. We also define a ***pruning round*** as a sequence of consecutive pruning iterations, which includes a successful iteration and any unsuccessful iterations right before it.

IV. Evaluation

A. Datasets

We examine three datasets from existing research [7], [11], [12]. These datasets consists of over 3 million power and EM traces acquired from microcontrollers and FPGAs, including the AVR XMEGA (8-bit), AVR ATMEGA (8-bit), ARM STM32F (32-bit), and AMD/Xilinx Artix-7 FPGA. Microcontrollers run unmasked or masked software implementation of AES written in C or assembly. FPGAs run a hardware implementation of AES written in Verilog. The details of the datasets, including the targets, the number of traces, and Points of Interest (POIs) can be found in Table I.

TinyPower Dataset. TinyPower dataset [12] consists of power and EM traces collected using ChipWhisperer with targets including the AVR XMEGA, ARM STM32, and Artix-7 FPGA. It includes multiple subsets. Each subset includes traces from two distinct targets of the same type for both same-device and cross-device evaluations, where the keys on the two targets are different. Additionally, TinyPower dataset includes subsets consisting of traces with random delays, where the samples in each trace are randomly delayed by a number of r samples and $r \in [0, 50]$. For microcontrollers, the POIs are identified as samples from the `SubBytes` of the first round of AES-128. Specifically, POIs are [1800, 2800] for XMEGA and [1200, 2200] for STM32. For FPGA, the POIs are [0, 100], which are samples from the entire AES-128 execution. Each subset in the TinyPower dataset is named in the format of `Target_Key_NumberOfTraces` [12].

ASCADv1 Dataset. ASCADv1 dataset [7] provides 100,000 power traces acquired from an AVR ATMEGA running a first-order masked software AES-128. The dataset consists of two subsets, one from a fixed key and one from multiple random keys. In our evaluation, we leverage the fixed key subset, which includes 50,000 training traces and 10,000 test traces. The POI is pre-selected as [45400, 46100], which consists of samples associated with the 3rd byte of `SubBytes` from the first round of AES-128. We report attack results in the same-device setting only over the ASCADv1 dataset as there are no traces provided from another device.

ASCADv2 Dataset. ASCADv2 dataset [11] consists of 800,000 power traces collected from an ARM STM32F microcontroller running a third-order affine-masked AES-128 implementation with an additional shuffling countermeasure. As the raw data composes over 800 GB, the authors provide an extracted dataset consisting of the concatenation of POIs from the mask shares [205000, 210000] and the `SubBytes` from the first round [455000, 465000]. In total, a POI window of 15000 samples per trace is used for side-channel analysis. Similarly to the ASCADv1 dataset, we report attack results in the same-device setting only over the ASCADv2 dataset as no traces are provided for cross-device evaluations.

B. Experiment Setting and Neural Network Architectures

Our experiments are mainly conducted on an Ubuntu 24.04 machine with an Intel i9 14900K, 128GB memory, and

TABLE II: Hyperparameters of Baseline CNN [7]

Layer	Hyperparameters
Conv_1 to Conv_5	Filters: {64, 128, 256, 512, 512}, Size: 11, Activation: ReLu + AveragePooling, Pool: 2, Strides: 2
Dense_1	Units: 4096, Activation: ReLU
Dense_2	Units: 4096, Activation: ReLU
Output	Units: 256, Activation: softmax

TABLE III: Hyperparameters of Baseline ResNets [11]

Layer	Hyperparameters
Input	Input shape: (15000, 1)
Initial Conv_1	Filters: 16, Kernel size: 11, ReLU,
Residual Blocks (x9)	Conv – Filters: {16, 32, 64, 128, 256,... 256}, Size: 11, ReLU, Strides: 1, 2, ... 2 BatchNormalization Conv – Filters: {16, 32, 64, 128, 256,... 256}, Size: 11, Activation: None, Strides: 1
Prediction Branches	Units: 1024, Activation: ReLu (x34 for v1, x18 for v2) BatchNormalization
Output	Units: 256 – {alpha, beta, 16 * sbox_preds}, (Units: 16 – 16 * {index_pred} if v1), softmax

NVIDIA RTX 4090. We leverage the Identity (ID) model as the leakage model in all experiments as our preliminary results suggest that the ID model derives better results (i.e. lower MTD) than the Hamming Weight model over most the datasets.

We investigate two baseline neural networks in our evaluation: the CNN utilized over the ASCADv1 dataset, and the ResNet examined over the ASCADv2 dataset. The baseline CNN [7] is a VGG-type architecture, consisting of 5 convolutional block layers and 2 dense layers. Each convolutional block includes a convolutional layer followed by an average pooling layer. It has been widely used in recent research of deep-learning side channel attacks. The entire model consists of about 54 million parameters. Hyperparameters of this CNN can be seen in Table II. This baseline CNN reports attack results over one byte. We utilize this CNN for attack results over all the datasets except the ASCADv2 dataset.

For the baseline ResNet, we explore both versions, referred to as MultiSCAv1 and MultiSCAv2, described in [11]. Both versions leverage Multi-Task Learning to attack all 16 key bytes of an AES encryption key simultaneously with a single neural network over the ASCADv2 dataset. MultiSCAv1 predicts all the 16 bytes, two mask values, and all 16 shuffling indices. It consists of 1 convolutional layer, 9 residual blocks, 1 average pooling layer, and 34 parallel prediction branches (for two masked values r_m, r_{out}, 16 key bytes k_1..., k_{16}, and 16 shuffling indices s_1..., s_{16}). Each prediction branch consists of one dense layer, a batch normalization layer, and a final softmax. On the other hand, MultiSCAv2 reveals all the 16 bytes and mask values without predicting the shuffling indices. It shares the same architecture as MultiSCAv1 excluding the 16 prediction branches for the 16 shuffling indices. Hyperparameters of the baseline ResNets are listed in Table III.

Training and Pruning Parameters. When we train a baseline CNN, we always train it with 150 epochs. When we train a baseline ResNet, we train for 20 epochs adhering to an early stop strategy. The number of training traces varies and depends on each specific dataset. In general, a more challenging dataset needs a greater number of training traces. We will specify the number of training traces for each dataset in our later experiments. The number of validation traces and the number of test traces are both fixed as 5,000 for the evaluation over every dataset. When we perform pruning, we always set the initial number of pruning epochs e_{init} as the same as the one used by the baseline architecture, i.e. 150 epochs for CNNs and 20 epochs for ResNets. We set the increment pruning epochs e_{inc} as 25 for CNNs and 10 for ResNets respectively. We set the maximum number of epochs e_{max} as 650 and 60 for CNNs and ResNets respectively. We also set the pruning rate decrement p_{dec} as 0.1 for both CNNs and ResNets. We use up to 500 and 60 epochs with an early stop in the tempering (the final fine-tuning) step for CNNs and ResNets, respectively.

We also need to choose two critical parameters, including the accuracy threshold acc_{thres} and initial pruning rate p_{init}, for our pruning. We discuss the selection of these two parameters in detail in our later experiments. For each dataset, we use the same training traces for the fine-tuning step. For instance, if we use 140k traces for training, we will leverage those 140k traces for the fine-tuning step as well.

For the TinyPower and ASCADv1 dataset, we report the attack results from one byte of the AES key. We select the 3rd byte for results as other bytes will be similar as demonstrated in existing studies [7], [8], [19]. For the ASCADv2 dataset, we report attack results from all the 16 bytes as the baseline ResNet reveals all key bytes simultaneously [11]. Given a trained neural network and a set of test traces, we always report the average MTDs over 100 iterations of testing, where the order of test traces is randomly shuffled in each iteration. For same-device evaluations, training traces and test traces are selected from the same dataset but do not overlap. For cross-device evaluations, we choose traces collected with a different device and different key for testing. For instance, given a neural network trained with S1_K1_200k, we leverage traces from S2_K2_100k for testing in cross-device evaluations.

C. Experiments

Experiment 1: Comparison between Baseline and Our Pruned Neural Networks. We first obtain attack results with the baseline CNNs and ResNets over the datasets we examine. As we can observe from Table IV and Table V, these baseline neural networks can effectively recover keys given each dataset and our results are consistent with the ones reported in recent studies [7], [12]. We also include the number of training traces for the evaluation over each dataset in the tables. The number of training traces for each dataset is recommended by previous studies. It is also worth mentioning that MultiSCAv2 derives higher MTD than the ones from MultiSCAv1. However, as mentioned in [11], it does not require knowing shuffling index values during the training phase compared to the MultiSCAv1. *While each baseline neural network recovers keys, the number of neural network parameters is extremely high.*

TABLE IV: Comparison between Baseline and Our Pruned CNNs (Initial Pruning Ratio: 0.3, Accuracy Threshold: 0.006 for F1_K1_200k and ASCADv1 and 0.008 for others datasets)

Target	Training Dataset	No. of Training/ Pruning Traces	CNN	MTD (3rd Byte) Same-Device	MTD (3rd Byte) Cross-Device	Parameters	Reduction Rate (%)	Training/Pruning Time (s)
XMEGA	X1_K1_200k	50k	Baseline	2	3	54,069,632	–	712
			Pruned	13	12	**1,044**	**99.998**	**6,659**
	X1_K1_150k_EM	50k	Baseline	2	25	54,069,632	–	704
			Pruned	6	75	**2,579**	**99.995**	**12,459**
	X1_K1_200k_RD	50k	Baseline	2	2	54,069,632	–	722
			Pruned	5	7	**2,724**	**99.995**	**14,605**
STM32F	S1_K1_200k	50k	Baseline	2	3	54,069,632	–	710
			Pruned	5	19	**3,126**	**99.994**	**20,936**
	S1_K1_150k_EM	140k	Basline	25	80	54,069,632	–	1,963
			Pruned	34	130	**3,059**	**99.994**	**42,170**
	S1_K1_200k_RD	50k	Baseline	6	7	54,069,632	–	706
			Pruned	17	15	**1,044**	**99.998**	**29,378**
Artix-7	F1_K1_200k	180k	Baseline	439	2,614	24,709,504	–	1,050
			Pruned	346	3,086	**98,711**	**99.60**	**118,505**
ATMEGA	ASCADv1	50k	Baseline	356	NA	43,583,872	–	550
			Pruned	321	NA	**62,136**	**99.86**	**30,837**

TABLE V: Comparison between Baseline and Our Pruned ResNets over ASCADv2 dataset (STM32F, No. of Training/Pruning Traces: 420k, B: Baseline; P: Pruned; RR: Reduction Rate, T/P Time: Training/Pruning Time, Initial Pruning Rate: 0.3, Accuracy Threshold: 0.006)

	MTD across all the 16 bytes (Same-Device Only)																Parameters	RR(%)	T/P Time (s)
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th	12th	13th	14th	15th	16th			
v1(B)	56	79	72	65	64	71	73	63	78	72	69	67	68	64	72	73	137,471,808	–	15,351
v1(P)	223	253	254	291	237	368	208	309	314	519	334	239	218	286	320	282	**11,349,292**	**91.74%**	**196,048**
v2(B)	140	382	268	211	134	362	225	194	195	529	382	668	127	246	199	142	78,407,232	–	12,365
v2(P)	268	379	512	537	200	487	295	436	535	527	433	278	263	394	393	176	**8,502,342**	**89.16%**	**120,371**

Next, we perform our iterative pruning algorithm on each baseline CNN or ResNet. During our pruning in this experiment, we choose initial pruning rate p_{init} as 0.3 for all the datasets and accuracy threshold acc_{thre} as 0.008 for all the datasets except F1_K1_200k, ASCADv1 and ASCADv2. We choose accuracy threshold acc_{thre} as 0.006 for F1_K1_200k, ASCADv1 and ASCADv2. We defer the discussions on the selection of these two pruning parameters in later experiments.

As presented in Table IV and V, our pruning can successfully derive an extremely high parameter reduction rate and the pruned neural network can still recover keys effectively over every dataset we examine in both the same-device setting and cross-device setting (when applicable). Compared to the MTDs obtained in baseline neural networks, the ones derived by our pruned neural networks, in general, only increase slightly. *In other words, our pruning algorithm can significantly reduce the number of parameters in a neural network with a minor impact on the results of side-channel attacks.* As a necessary tradeoff, our pruning algorithm requires longer time than simply training a baseline neural network.

For instance, given the ASCADv1 dataset, our pruned neural network achieves a parameter reduction rate of 99.86% (62,136 v.s. 43.58 million) and can still recover keys with a similar number of test traces (321 v.s. 356) compared to its baseline CNN. Given the ASCADv2 dataset, our pruned ResNet can achieve 89.16% parameter reduction rate (8.50 million v.s. 78.41 million) on MultiSCAv2 and can still recover keys over all the 16 key bytes. *We also observe that our pruning algorithm achieves a lower reduction rate over traces from masked AES.* This is expected as traces from masked AES carry less side-channel leakage and are more challenging to attack compared to traces acquired from unmasked AES.

Experiment 2: The Impact of Accuracy Threshold. We investigate the impacts of accuracy threshold on attack results, parameter reduction rate, and the overall pruning time. Specifically, we perform our iterative pruning over four datasets, including S1_K1_150k_EM, S1_K1_200K_RD, F1_K1_200k, and ASCADv1, with accuracy thresholds $\{0.006, 0.008, 0.010, 0.012\}$ respectively. We select these four datasets as they are relatively representative; covering various scenarios with traces from EM, random delay, FPGAs, and masked AES. For each pruning process, given each dataset's own baseline CNN reported in Table IV, we set the initial pruning rate p_{init} as 0.5 in this experiment. We summarize our pruning results in Table VI. It is also worth mentioning that, according to our experience, we should avoid selecting the accuracy threshold lower than 0.006 as it would be too close to random guess (i.e. 1/256 $\approx$ 0.0039 given ID model) and a neural network achieving an accuracy similar to random guess, typically, fails to distinguish correct keys.

First, we observe the selection of accuracy threshold is *extremely sensitive* for F1_K1_200k and ASCADv1 (consisting of traces collected from FPGAs or masked AES), where only an accuracy threshold of 0.006 can prune a CNN successfully. When we increase it to 0.008 or higher, our pruning fails to derive a pruned CNN for both datasets (i.e. our pruning still runs but no parameters are removed). In essence, this is because the two datasets carry much lower leakage and achieving an accuracy greater or equal to 0.008 is infeasible during the fine-tuning process (as with the baseline models). The pruning time

TABLE VI: The Impact of Accuracy Threshold ($\perp$: failing to derive a pruned neural network; Initial Pruning Rate: 0.5)

Target	Training Dataset	Accuracy Threshold	MTD (3rd Byte) Same-Device	MTD (3rd Byte) Cross-Device	Parameters	Reduction Rate (%)	Pruning Time (s)
STM32F	S1_K1_150k_EM	0.006	36	175	55,227	99.90	104,759
		0.008	**30**	**129**	**2,769**	**99.995**	**78,733**
		0.010	47	218	2,501	99.995	79,978
		0.012	34	105	3,126	99.994	94,041
	S1_K1_200K_RD	0.006	9	11	3,059	99.994	31,402
		0.008	**4**	**4**	**2,691**	**99.995**	**27,930**
		0.010	4	5	2,613	99.995	32,874
		0.012	16	15	2,049	99.996	34,691
Artix-7	F1_K1_200k	**0.006**	**342**	**2,580**	**218,424**	**99.12**	**141,790**
		0.008	$\perp$	$\perp$	24,709,504	$\perp$	165,459
ATMEGA	ASCADv1	**0.006**	**229**	NA	**11,878**	**99.97**	**48,943**
		0.008	$\perp$	$\perp$	43,583,872	$\perp$	67,852

TABLE VII: The Impact of Initial Pruning Rate (Accuracy threshold: 0.006 for S1_K1_150k_EM and S1_K1_200k_RD and 0.008 for F1_K1_200k and ASCADv1)

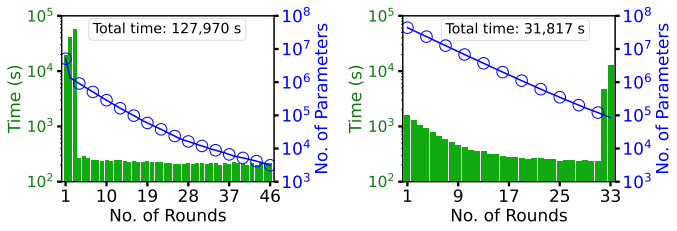
Target	Training Dataset	Initial Pruning Rate	MTD (3rd Byte) Same-Device	MTD (3rd Byte) Cross-Device	Parameters	Reduction Rate (%)	Pruning Time (s)
STM32F	S1_K1_150k_EM	0.1	45	352	101,876	99.812	32,692
		0.3	**34**	**130**	**3,059**	**99.994**	**42,170**
		0.5	74	659	2,613	99.995	83,939
		0.7	31	63	3,126	99.994	128,790
	S1_K1_200K_RD	0.1	21	16	15,126	99.972	14,965
		0.3	**17**	**15**	**1,044**	**99.998**	**29,378**
		0.5	5	11	3,126	99.994	30,986
		0.7	21	23	1,337	99.998	45,375
Artix-7	F1_K1_200k	0.1	455	2,325	575,053	97.67	34,849
		0.3	**346**	**3,086**	**98,711**	**99.60**	**118,505**
		0.5	362	2,509	322,829	98.69	172,859
		0.7	330	3,833	2,153,374	91.29	154,519
ATMEGA	ASCADv1	0.1	2,376	NA	19,033,834	56.33	16,654
		0.3	**321**	NA	**62,136**	**99.86**	**30,837**
		0.5	236	NA	24,135	99.94	37,149
		0.7	1,087	NA	39,377	99.91	42,590

for F1_K1_200k dataset is longer than the one from ASCADv1 dataset as it requires a much greater number of traces in the fine-tuning process (180k vs. 50k). *Based on the observation, we set the accuracy threshold as 0.006 for all the datasets from FPGAs or masked AES, including F1_K1_200k, ASCADv1, and ASCADv2, in our remaining evaluations.*

Second, we find that for less challenging datasets, such as S1_K1_150k_EM and S1_K1_200k_RD, all the selected accuracy thresholds we examine are able to derive a pruned CNN, which can recover keys effectively and achieve extremely high parameter reduction rates. This is expected as the accuracy of the baseline CNN can achieve accuracy as high as 0.02 (i.e., 2%). The accuracy thresholds we select in this experiment are still sufficiently higher than random guess but lower than the accuracy of the baseline CNN. Between the two datasets, S1_K1_150k_EM and S1_K1_200k_RD, we notice that the pruning time is longer on the S1_K1_150k_EM dataset. This is mainly because it requires a greater number of traces during the fine-tuning process (140k vs. 50k). *Overall, there is a minor impact on the performance (MTDs, parameter reduction rate, and pruning time) of a pruned CNN given the accuracy threshold values we examined.* When choosing 0.008, it requires a slightly shorter pruning time than the others. *Therefore, for all the remaining experiments over datasets collected from STM32F (and XMEGA) running unmasked AES, we select 0.008 as the accuracy threshold.*

Experiment 3: The Impact of Initial Pruning Rate. We experimentally investigate the impact of initial pruning rate p_{init} on the performance of our pruning algorithm. Specifically, we fix all the other pruning parameters and perform our pruning over four datasets, including S1_K1_150k_EM, S1_K1_200K_RD, F1_K1_200k, and ASCADv1, with initial pruning rates of $\{0.1, 0.3, 0.5, 0.7\}$ respectively. We summarize our results in Table VII and have three main observations.

First, given each of the initial pruning rates we provide, our pruning algorithm can eventually obtain a pruned neural network that can still effectively recover keys in the same-device setting and the cross-device setting (when applicable). Second, the overall pruning time, in general, increases significantly when the initial pruning rate increases across all the four datasets. For instance, given the ASCADv1 dataset, when the initial pruning rate is 0.1, the entire pruning time is only 16,654 seconds (4.63 hours). However, when we choose pruning rate as 0.7, the entire pruning time jumps to 42,590 seconds (11.83 hours). Third, the parameter reduction rate is the highest when the initial pruning rate is 0.3 or 0.5 given each of the four datasets. In other words, providing an extremely low initial pruning rate (0.1) or an extremely high initial pruning rate (0.7) does not lead to a minimal (or nearly minimal) number of parameters in a neural network, especially given



(a) Initial pruning rate as 0.7 (b) Iinitial pruning rate as 0.1

Fig. 3: Comparison of our pruning time per round over S1_K1_150k_EM given different initial pruning rates.

more challenging datasets (F1_K1_200k and ASCADv1).

The above observation on pruning time is mainly because a greater initial pruning rate prunes a baseline neural network aggressively in the first round, which leads to more iterations searching for a child network to meet a given accuracy threshold. In addition, a greater initial pruning ratio leads to a greater number of rounds in our pruning, which also increases the total pruning time. A further breakdown of the pruning time per round from one dataset S1_K1_150k_EM is presented in Fig. 3a given the initial pruning rate is large (e.g., 0.7). The pruning spends a much longer time in the first few rounds and requires 46 rounds in total to complete the entire pruning.

On the other hand, a smaller initial pruning rate prunes a baseline neural network less aggressively in the first round, which allows a child network to be pruned quickly with less need on updating pruning parameters. As a result, our pruning algorithm can move to the next round quickly and leads to a lower number of rounds overall, which reduces the entire pruning time. A further breakdown of the pruning time per round from one dataset S1_K1_150k_EM is presented in Fig. 3b given the initial pruning ratio is small (e.g., 0.1). The pruning spends less time in the first few rounds and only needs 33 rounds in total to finish the entire pruning.

Based on our observations, we believe that both 0.3 and 0.5 are good selections as the initial pruning rate. We select the initial pruning rate as 0.3 in the rest of our experiments as it requires less overall pruning time than the one with 0.5.

Experiment 5: Variance of Our Pruning Results. As our pruning algorithm involves multiple iterations of pruning and fine-tuning neural network parameters, the pruned neural networks we obtain are naturally non-deterministic. In other words, running our pruning algorithm twice will derive two distinct pruned neural networks, even given the same set of pruning parameters and the same dataset.

In this experiment, we run our pruning algorithm with multiple trials on the same dataset with the same set of pruning parameters to investigate variances on MTDs, parameter reduction rates, and pruning time. Specifically, we repeat our algorithm over four trials on ASCADv1 dataset with accuracy threshold 0.006 and initial pruning rate 0.3. In addition, we also repeat our algorithm four trails on S1_K1_200k_RD with accuracy threshold 0.008 and initial pruning rate 0.3. The detailed results are summarized in Table VIII.

TABLE VIII: Variance of Our Algorithm across Multiple Trials

Training Dataset	Trial	MTD (3rd Byte) Same	MTD (3rd Byte) Cross	Reduction Rate (%)	Pruning Time (s)
S1_K1_200k_RD	1	17	15	99.998	29,378
	2	11	10	99.998	14,167
	3	19	23	99.994	16,970
	4	6	6	99.974	29,544
ASCADv1	1	321	NA	99.86	30,837
	2	213	NA	99.54	16,508
	3	247	NA	99.73	16,346
	4	797	NA	99.84	25,269

TABLE IX: Comparison Among the Smallest (but Still Effective) CNNs Obtained by Different Methods (Network Search: [13], Single-Shot Pruning: [12], Target: STM32, $\perp$: failing to converge to a key within 5,000 traces)

Training Dataset	Method	MTD (3rd Byte) Same	MTD (3rd Byte) Cross	Parameters	Pruning/Search Time (hrs)
S1_K1_200k_RD	[13]	3,462	$\perp$	**149**	22.01
	[12]	**4**	**5**	3.75×10^6	**0.27**
	Ours	11	10	1,044	3.94
S1_K1_150k_EM	[13]	2,577	$\perp$	**567**	52.11
	[12]	**31**	**33**	4.85×10^6	**0.72**
	Ours	34	130	3,059	11.71
ASCADv1	[13]	4,207	NA	**753**	32.23
	[12]	**307**	NA	2.13×10^6	**0.30**
	Ours	321	NA	62,136	8.57

We can observe that variances of MTDs, parameter reduction rate, and pruning time on S1_K1_200k_RD are minor. On the other hand, the variances of MTDs, parameter reduction rate, and pruning time on ASCADv1 are slightly higher. These results indicate that our iterative pruning algorithm derives higher variances given a more challenging target.

Experiment 6: Comparison with Previous Studies. We compare our iterative pruning with two existing studies, including single-shot pruning in [12] and neural network architecture search [13]. Li et al. [12] adopted single-shot pruning, where a neural network is pruned with a single iteration but each layer is pruned with a customized pruning rate. The pruning rate is automatically derived by their algorithm based on the distributions of the filter scores at each layer. We run their source code and utilize l_2 norm as the score algorithm. Rijsdijk et al. [13] applied network architecture search for side-channel attacks by leveraging *reinforcement learning* to search the optimal neural network architecture given a dataset. We run their source code [13] over our datasets to find small CNNs to compare the ones we obtained from our pruning. Specifically, we tune the hyperparameters of a CNN using *Reward Small* setting described in [13] and search for up to 500 CNNs given each dataset, where each CNN is trained with 150 epochs. We also utilize Hamming Weight model rather than ID model in this neural network architecture search as the HW model derives lower MTDs. We adopt the same hyperparameter search space as the one in [13].

As shown in Table IX, both ours and neural network architecture search [13] derive extremely small neural networks. However, our algorithm requires less pruning/search time (3.7∼5.5X faster) and derives neural networks with much lower MTDs than neural network architecture search in both

TABLE X: Compatibility of Our Pruning with Quantization (P: Pruned; Q: Quantized)

Training Dataset	CNN	MTD (3rd Byte) Same	MTD (3rd Byte) Cross	CNN Size (KBs)
S1_K1_200k_RD	P	11	10	82.77
	P&Q	12	13	**14.67**
S1_K1_150k_EM	P	34	130	104.49
	P&Q	35	140	**18.88**
F1_K1_200k	P	346	3,086	847.23
	P&Q	466	2,472	**113.88**
ASCADv1	P	321	NA	561.34
	P&Q	351	NA	**75.84**

same-device and cross-device settings. Single-shot pruning [12] requires a much shorter pruning time than the other two methods, but the neural networks generated by single-shot pruning still carry millions of parameters.

Experiment 7: Compatibility with Quantization. Quantization, which reduces the number of bits of parameters, is another common technique to compress the size of neural network, either independently or after pruning is applied. We show that our pruning algorithm is compatible with existing quantization methods, where our pruned neural networks can be further reduced in size due to quantization but still recover keys effectively. Note that quantization, in general, does not change the number of parameters in a neural network. Specifically, we leverage Tensorflow's TFLite *dynamic range quantization algorithm* [20], where the weights of a pruned neural network are quantized from 32 bits to 8 bits. As shown in Table X, after applying quantization to our pruned neural networks, MTDs increase slightly but the size of each neural network is further reduced by a factor of 5.5$\sim$7.4.

Experiment 8: Pruned Neural Networks on Embedded Devices. We demonstrate that our pruned neural networks can perform side-channel attacks efficiently and effectively on embedded devices, including a NVIDIA Jetson Orin Nano and an AMD/Xilinx Zynq ZCU104 Evaluation Board. The Jetson consists of a NVIDIA Ampere GPU, a 6-core ARM CPU, and 8GB memory. The ZCU104 FPGA board is equipped with a quad-core ARM Cortex-A53 applications processor, dual-core Cortex-R5 real-time processor, Mali-400 MP2 graphics processing unit, and 16nm FinFET+ programmable logic.

For the NVIDIA Jetson Orin Nano, we first train baseline networks and obtain pruned neural networks on our Nvidia 4090 GPU machine. Next, we run both baseline models and our pruned neural networks on Jetson to primarily compare their size, memory usage, and inference time. As shown in Table XI, our pruned neural networks require significantly lower storage and memory usage compared to baseline networks on Jetson. The inference time per trace of our pruned neural networks are slightly lower than baseline neural networks.

To be able to compile and run neural networks on the ZCU104 FPGA board, we use the Xilinx Vitis AI framework [21]. As a result, we have to make the following changes in our experiment pipeline, including (1) updating 1dCNN layers to 2dCNN layers in the baseline CNN (this does not change the number of parameters); (2) scaling samples in each trace

TABLE XI: Comparison between Baseline and Our Pruned CNNs/ResNets on Nvidia Jetson (B: Baseline; P: Pruned)

Training Dataset	CNN/ ResNet(v2)	Size (MBs)	Memory (MBs)	Attack Time Per Trace (ms)
S1_K1_200k_RD	B	412.59	501.57	5.96
	P	**0.08**	**12.43**	5.95
S1_K1_150k_EM	B	412.59	501.55	2.16
	P	**0.10**	**12.68**	1.87
F1_K1_200k	B	188.59	310.43	1.01
	P	**0.83**	**16.18**	0.97
ASCADv1	B	332.59	410.55	1.12
	P	**0.55**	**17.17**	0.98
ASCADv2	B	897.84	1,606.61	2.37
	P	**98.10**	**210.35**	1.70

TABLE XII: Comparison between Baseline and Our Pruned CNNs on AMD/Xilinx ZCU104 FPGA Board (B: Baseline; P: Pruned; Q: Quantized; C: Compiled)

Training Dataset	CNN	MTD (3rd Byte) Same	MTD (3rd Byte) Cross	Size (MBs)	Attack Time Per Trace (ms)
S1_K1_200k_RD	B	1	371	412.59	0.21 [GPU]
	P	7	35	0.09	0.16 [GPU]
	P&Q	17	38	0.15	0.34 [GPU]
	P&Q&C	14	48	0.19	0.49 [FPGA]
S1_K1_150k_EM	B	18	272	412.59	0.28 [GPU]
	P	79	256	0.08	0.16 [GPU]
	P&Q	87	276	0.14	0.34 [GPU]
	P&Q&C	101	298	0.19	0.48 [FPGA]
F1_K1_200k	B	847	2,582	188.59	0.22 [GPU]
	P	803	2,738	0.31	0.16 [GPU]
	P&Q	916	3,854	0.26	0.33 [GPU]
	P&Q&C	947	3,781	0.20	0.37 [FPGA]
ASCADv1	B	488	NA	332.59	0.25 [GPU]
	P	266	NA	0.57	0.15 [GPU]
	P&Q	358	NA	0.38	0.34 [GPU]
	P&Q&C	309	NA	0.24	0.45 [FPGA]

in advance from `float32` to `int8` format (i.e., quantizing each sample from 32 bits to 8 bits). Given these changes, we first train and prune a baseline CNN on the Nvidia 4090 GPU machine over quantized traces with our iterative pruning. Given the pruned CNN, we then perform quantization (*power of 2 scale quantization*, `pos2`, a post-training quantization provided by the Vitis-AI framework) over the pruned CNN and compile the pruned & quantized CNN with the Xilinx Vitis-AI framework. Finally, the compiled CNN is uploaded to the ZCU104 FPGA board, where all the operations before the last layer are done within the FPGA and the operations of the last layer are computed within the CPU. The splitting of the operations is done automatically by Vitis AI. We then run the calculation of key ranks with the CPU on the ZCU104 board.

We compare the MTDs, the size of neural networks, the attack time per trace across multiple stages of generating a neural network in Table XII. We find that the compiled CNN on the FPGA, in general, can effectively recover keys with a slightly higher MTDs than the pruned one. This is likely because quantization and compilation process moderately reduce the performance of neural networks. For S1_K1_RD and S1_K1_EM, the size of the compiled neural network increases slightly compared the pruned one as Vitis AI adds additional meta data to the files uploaded to FPGAs. The number of parameters in the compiled neural network remains the same compared to the pruned one. The attack time per trace on the FPGA is slightly slower than the on GPUs but still highly

efficient with less than 0.49 milliseconds per trace.

V. Discussions and Limitations

Micro Neural Networks for Side-Channel Analysis. *From the perspective of an adversary*, being able to effectively perform side-channel attacks with pre-trained micro neural networks running on embedded devices would allow this attacker to recover keys in a more stealthy approach rather than forwarding and processing traces on machines with massive GPUs. On the other hand, *from the perspective of a security analyzer*, being able to operate tiny neural networks on embedded devices to analyze side-channel information could facilitate on-device verification to monitor potential key leakage and benefit side-channel-based on-device detection against malicious behaviors, such as malware [22], [23], hardware Trojans [24], [25], adversarial examples [26], and side-channel attacks [27].

Optimization on Pruning Time. Compared to single-shot pruning, one major tradeoff in our iterative pruning is the longer pruning time. There are ways to further optimize the total pruning time. For instance, increasing increment number of epochs and lowering the maximal number of epochs can reduce the number of iterations and optimize total pruning time. In addition, introducing early stop to halt the iterative pruning when multiple consecutive unsuccessful iterations happen could also reduce the total pruning time. Due to space limitation, we are not able to examine these aspects and their impacts on attack results and parameter reduction rate.

VI. Related Works

Pruning for SCA. Perin et al. [28] apply unstructured pruning to reduce neural network size for side-channel attacks by leveraging the Lottery Ticket Hypothesis. Unstructured pruning sets less important weights in a neural network as zeros rather than removing them. As a result, this method does not reduce model memory usage during the attack. A recent study by Li et al. [12] investigates single-shot structured pruning to reduce the size of neural networks for side-channel attacks by removing less important filters. They examine two score algorithms, including l_2 norm and FPGM [29], and design an algorithm named MiniDrop to automatically determine a customized pruning rate for each layer based on filter scores. They observe that there are no significant difference between l_2 norm and FPGM in terms of parameter reduction rate.

Hyperparameter/Architecture Search for SCA. Several recent works [8], [30], [13], [31] have attempted to create small neural networks directly using hyperparameter or architecture search rather than pruning a pre-trained neural network. Zaid et al. [8] apply neural network virtualization methods to obtain hyperparameters needed for side-channel attacks. For instance, they obtain a small CNN with 87,279 parameters and this CNN recovers keys with 244 test traces given the ASCADv1 dataset with a random delay up to 50. This work is extended by Wouters et al. in [30], which can further reduce the number of parameters from the model with additional trace pre-processing. They derive a smaller CNN with only 41,052 parameters (but no reports on MTDs) given ASCADv1 dataset with a random delay up to 50. However, both methods require expert-in-the-loop to manually examine/visualize parameter importance and the time that is needed to search for these small neural networks is not reported. Acharya et. al. [32] leverage neural network search and information theoretic metrics to guide the search of hyperparameters, the number of training epochs, and the number of training traces.

Other studies apply automatic hyperparameter search methods, including reinforcement learning [13], Bayesian parameter optimization [31], and budget-based Bayesian optimization with hyperband search [33] to seek small neural networks. Wu et al. [31] explore Bayesian optimization for neural network architecture search. In essence, they run Bayesian optimization to update hyperparameters through multiple iterations of neural network training, where the neural network with the best attack performance is selected at the end. A CNN with 1.08 million parameters over ASCADv1 dataset is reported. Yap et al. [33] further extend Bayesian optimization but adopt a budget scheme to bound the exhaustive hyperparameter search.

Other Aspects of SCA. Besides optimizing the size of neural networks, the portability of neural networks is another critical aspect of deep learning side-channel attacks extensively examined in existing research [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], where domain shifts exist between training and test data due to hardware, software, and data acquisition variations. Other aspects, such as explainability [44], [45], selection of POIs [46], [47], limited number of traces [48], [49], leakage model [50], imbalanced data [51], scheme-aware modeling [52], more complex neural networks [19], and pre-silicon side-channel analysis with neural networks [53], [54], [55] have also been investigated. Non-profiling attacks with neural networks are feasible as shown in [14], [56], [57], [58], [59]. More comprehensive surveys on deep learning side-channel attacks can be in [9], [10].

Recent research also investigate SCA in general, such as the combination of power and EM channels [60], weakly profiling attacks [61], and automatic leakage model selection [62], to optimize attack performance. Design tools for side-channel-aware implementations are discussed in [63]. Recent studies of utilizing SCA to reveal architectures and weights of neural networks are summarized in [64].

VII. Conclusion

We propose an iterative pruning algorithm to optimize the size and memory usage of neural networks for side-channel attacks. Experimental results show that our algorithm is effective over existing datasets. We also demonstrate it is feasible to operate these micro neural networks on embedded devices.

Acknowledgments

The authors thank the anonymous reviewers for their suggestions. This work was supported in part by National Science Foundation (CNS-2150086, CNS-2212010, DGE-2043106) and CHEST — NSF IUCRC Center for Hardware and Embedded System Security and Trust (CNS-1916722, CNS-1916762).

REFERENCES

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Proc. of CRYPTO'99*, 1999.

[2] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *Cryptographic Hardware and Embeeded Systems (CHES'04)*, 2004.

[3] S. Chari, J. R. Rao, and P. Rohatgi, "Template Attacks," in *Cryptographic Hardware and Embeeded Systems (CHES'02)*, 2002.

[4] W. Schindler, K. Lemke, and C. Paar, "A stochastic model for differential side channel cryptanalysis," in *Cryptographic Hardware and Embedded Systems (CHES'05)*, 2005.

[5] H. Maghrebi, T. Portigliatti, and E. Proff, "Breaking cryptographic implementations using deep learning techniques," in *Proc. of International Conference on Security, Privacy and Applied Cryptography Engineering (SPACE'16)*, 2016.

[6] E. Cagli, C. Dumas, and E. Prouff, "Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures," in *Proc. of CHES'17*, 2017.

[7] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ASCAD database," *Journal of Cryptographic Engineering*, vol. 10, no. 2, 2020.

[8] G. Zaid, L. Bossuet, H. A, and A. Venelli, "Methodology for Efficient CNN Architectures in Profiling Attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[9] M. Panoff, H. Yu, H. Shan, and Y. Jin, "A Review and Comparison of AI-enhanced Side Channel Analysis," *J. Emerg. Technol. Comput. Syst.*, 2022.

[10] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina, "SoK: Deep Learning-based Physical Side-channel Analysis," *ACM Computing Surveys*, vol. 55, no. 11, 2023.

[11] L. Masure and R. Strullu, "Side channel analysis against the ANSSI's protected AES implementation on ARM," 2021. [Online]. Available: https://eprint.iacr.org/2021/592

[12] H. Li, M. Ninan, B. Wang, and J. M. Emmert, "TinyPower: Side-Channel Attacks with Tiny Neural Networks," in *Proc. of HOST'24*, 2024.

[13] J. Rijsdijk, L. Wu, G. Perin, and S. Picek, "Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.

[14] B. Timon, "Non-Profiled Deep Learning-based Side-Channel Attacks with Sensitivity Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 2, pp. 107–131, 2019.

[15] F. Standaert, T. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Proc. of EUROCRYPT'09*, 2009.

[16] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and Quantization for Deep Neural Network Acceleration: A Survey," in *Neurocomputing*, vol. 461. Elsevier, 2021.

[17] H. Cheng, M. Zhang, and J. Shi, "A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[18] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," in *Proc. of ICLR 2017*, 2017.

[19] S. Hajra, S. Chowdhury, and D. Mukhopadhyay, "EstraNet: An Efficient Shift-Invariant Transformer Network for Side-Channel Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024.

[20] TensorFlow, "Dynamic range quantization." [Online]. Available: https://ai.google.dev/edge/litert/models/post_training_quantization

[21] "Vitis-AI." [Online]. Available: https://github.com/Xilinx/Vitis-AI

[22] H. Sayadi, Y. Gao, H. M. Makrani, J. Lin, P. C. Costa, S. Rafatirad, and H. Homayoun, "Towards Accurate Run-Time Hardware-Assisted Stealthy Malware Detection: A Lightwegith yet Effectuve Time Series CNN-Based Approach," *Crypography*, vol. 5, no. 4, 2021.

[23] A. Kuruvila, S. Karmakar, and K. Basu, "Time Series-based Malware Detection using Hardware Performance Counters," in *Proc. of IEEE HOST'22*, 2022.

[24] K. I. Gubbi, B. S. Latibari, A. Srikanth, T. Sheaves, S. A. Beheshti-Shirazi, S. M. PD, S. Rafatirad, A. Sasan, H. Homayoun, and S. Salehi, "Hardware Trojan Detection Using Machine Learning: A Tutorial," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 3, 2023.

[25] T. Goruousis, Z. Zhang, M. Yan, M. Zhang, A. Mittal, A. Shrivastava, F. Restuccia, Y. Fei, and M. Onabajo, "Identification of Stealthy Hardware Trojans through On-Chip Temperature Sensing and an Autoencoder-Based Machine Learning Algorithm," in *IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2023.

[26] R. Ding, C. Gongye, S. Wang, A. Ding, and Y. Fei, "EMShepherd: Detecting Adversarial Samples via Side-Channel Leakage," in *Proc. of ACM ASIACCS'23*, 2023.

[27] Z. Pan and P. Mishra, "Automated Detection of Spectre and Meltdoan Attacks using Explainable Machine Learning," in *Proc. of IEEE HOST'22*, 2022.

[28] G. Perin, L. Wu, and S. Picek, "Gambling for Success: The Lottery Ticket Hypothesis in Deep Learning-Based Side-Channel Analysis," *Artificial Intelligence for Cybersecurity (Springer)*, 2022.

[29] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration," in *Proc. of CVPR'19*, 2019.

[30] L. Wouters, V. Arribas, B. Gierlichs, and B. Preneel, "Revisting a Methodology for Efficient CNN Architectures in Profiling Attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[31] L. Wu, G. Perin, and S. Picek, "I Choose You: Automated Hyperparameter Tuning for Deep Learning-Based Side-Channel Analysis," *IEEE Transactions on Emerging Topics in Computing*, vol. 12, no. 2, 2024.

[32] R. Y. Acharya, F. Ganji, and D. Forte, "Information Theory-based Evolution of NeuralNetworks for Side-channel Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[33] T. H. E. Yap, S. Bhasin, and L. Weissbart, "Train Wisely: Multifidelity Bayesian Optimization Hyperparameter Tuning in Side-Channel Analysis," in *Proc. of Selected Areas in Cryptography (SAC'24)*, 2024.

[34] D. Das, A. Golder, J. Danial, S. Ghosh, A. Raychowdhury, and S. Sen, "X-DeepSCA: Cross-Device Deep Learning Side Channel Attack," in *Proc. of 56th ACM/IEEE Design Automation Conference (DAC'19)*, 2019.

[35] A. Golder, D. Das, J. Danial, S. Ghosh, S. Sen, and A. Raychowdhury, "Practical Approaches Towards Deep-Learning Based Cross-Device Power Side Channel Attack," *IEEE. Trans. on Very Large-Scale Integration (VLSI) Systems*, vol. 27, no. 12, 2019.

[36] S. Bhasin, A. Chattopadhyay, A. Heuser, D. Jap, S. Picek, and R. R. Shrivastwa, "Mind the Portability: A Warriors Guide through Realistic Profiled Side-channel Analysis," in *Proc. of NDSS'20*, 2020.

[37] U. Rioja, L. Batina, and I. Armendariz, "When Similarities Among Devices are Taken for Granted: Another Look at Portability," in *Proc. of AFRICACRYPT 2020*, 2020, pp. 337 – 357.

[38] H. Yu, H. Shan, M. Panoff, and Y. Jin, "Cross-Device Profiled Side-Channel Attacks using Meta-Transfer Learning," in *Proc. of the 58th ACM/IEEE Design Automation Conference (DAC'21)*, 2021.

[39] J. Danial, D. Das, A. Golder, S. Ghosh, A. Raychowdhury, and S. Sen, "EM-X-DL: Efficient Cross-device Deep Learning Side-channel Attack with Noisy EM Signatures," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 18, no. 1, pp. 1–17, 2022.

[40] C. Wang, M. Ninan, S. Reilly, J. Ward, W. Hawkins, B. Wang, and J. M. Emmert, "Portability of Deep-Learning Side-Channel Attacks against Software Discrepancies," in *Proc. ACM WiSec'23*, 2023.

[41] H. Yu, S. Wang, H. Shan, M. Panoff, M. Lee, K. Yang, and Y. Jin, "DualLeak: Deep Unsupervised Active Learning for Cross-Device Profiled Side-Channel Leakage Analysis," in *Proc. of IEEE HOST'23*, 2023.

[42] M. Krcek and G. Perin, "Autoencoder-enabled Model Portability for Reducing Hyperparameter Tuning Efforts in Side-channel Analysis," *Journal of Cryptographic Engineering*, vol. 14, 2024.

[43] M. Ninan, E. Nimmo, S. Reilly, C. Smith, W. Sun, B. Wang, and J. M. Emmert, "A Second Look at the Portability of Deep Learning Side-Channel Attacks over EM Traces," in *Proc. of 27th International Symposium on Research in Attacks, Instrusions and Defenses (RAID'24)*, 2024.

[44] D. van der Valk, S. Picek, and S. Bhasin, "Kilroy Was Here: The First Step Towards Explainability of Neural Networks in Profiled Side-Channel Analysis," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2020.

[45] T. Yap, A. Benamira, S. Bhasin, and T. Peyrin, "Peek into the Black-Box: Interpretable Neural Network using SAT Equations in Side-Channel Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[46] T. Yap, S. Bhasin, and S. Picek, "OccPoIs: Points of Interest based on Neural Network's Key Recovery in Side-Channel Analysis through Occlusion," https://eprint.iacr.org/2023/1055.pdf.

[47] G. Perin, L. Wu, and S. Picek, "Exploring Feature Selection Scenarios for Deep Learning-based Side-channel Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022.

[48] C. Wang, J. Dani, S. Reilly, A. Brownfield, B. Wang, and J. M. Emmert, "TripletPower: Deep-Learning Side-Channel Attacks over Few Traces," in *Proc. of IEEE HOST'23*, 2023.

[49] L. Wu, L. Weissbart, M. Krcek, H. Li, G. Perin, L. Batina, and S. Picek, "Label Correlation in Deep Learning-Based Side-Channel Analysis," *IEEE Transactions on Information Forensics and Security*, 2023.

[50] L. Wu, A. Ali-Pour, A. Rezaeezade, G. Perin, and S. Picek, "Breaking Free: Leakage Model-free Deep Learning-based Side-channel Analysis," https://eprint.iacr.org/2023/1110.pdf.

[51] S. Picek, A. Heuser, A. Jovic, and F. Regazzoni, "The curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 1, pp. 209–237, 2019.

[52] L. Masure, V. Cristiani, M. Lecomte, and F.-X. Standaert, "Don't Learn What You Already Know Scheme-Aware Modeling for Profiling Side-Channel Analysis againstMasking," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[53] D. Shanmugam and P. Schaumont, "Improving Side-Channel Leakage Assessment Using Pre-Silicon Leakage," in *International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2023.

[54] L. Lin, D. Zhu, J. Wen, H. Chen, Y. Lu, N. Cheng, C. Chow, H. Shrivastav, C. W. Chen, K. Monta, and M. Nagata, "Multiphysics Simulation of EM Side-Channels from Silicon Backside with ML-based Auto-POI Identification," in *IEEE HOST'21*, 2021.

[55] A. Srivastava, S. Das, N. Choudhury, R. Psiakis, P. H. Silva, D. Pal, and K. Basu, "SCAR: Power Side-Channel Analysis at RTL Level," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 6, 2024.

[56] D. Kwon, H. Kim, and S. Hong, "Non-Profiled Deep Learning-based Side-Channel Preprocessing with Autoencoders," *IEEE Access*, 2021.

[57] L. Wu, S. Tiran, G. Perin, and S. Picek, "An End-to-end Plaintext-based Side-channel Collision Attack without Trace Segmentation," https://eprint.iacr.org/2023/1109.pdf.

[58] L. Wu, G. Perin, and S. Picek, "Hiding in Plain Sight: Non-profiling Deep Learning-based Side-channel Analysis with Plantext/Ciphertext," https://eprint.iacr.org/2023/209.pdf.

[59] M. Staib and A. Moradi, "Deep learning side-channel collision attack," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[60] Y. Bai, J. Park, M. Tehranipoor, and D. Forte, "Dual Channel EM/Power Attack Using Mutual Information and Its Real-time Implementation," in *Proc. of IEEE HOST'23*, 2023.

[61] L. Wu, G. Perin, and S. Picek, "Weakly Profiling Side-Channel Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024.

[62] O. Bazangani, A. IOOSS, I. Buhan, and L. Batina, "ABBY: Automating leakage modelling for side-channel analysis," in *Proc. of ACM AISACCS'24*, 2024.

[63] I. Buhan, L. Batina, Y. Yarom, and P. Schaumont, "SoK: Design Tools for Side-Channel-Aware Implementations," in *Proc. of ACM AISACCS'22*, 2022.

[64] P. Horvath, D. Lauret, Z. Liu, and L. Batina, "SoK: Neural Network Extraction Through Physical Side Channels," in *Proc. of USENIX Security'24*, 2024.