TinyPower: Side-Channel Attacks with Tiny Neural Networks

Haipeng Li, Mabon Ninan, Boyang Wang, John M. Emmert
University of Cincinnati

***Abstract*—Side-channel attacks leverage the correlation between power consumption and intermediate results of encryption to infer encryption keys. Recent studies show that deep learning offers promising results in the context of side-channel attacks. However, neural networks utilized in deep-learning side-channel attacks are often complex with substantial amounts of parameters and consume significant memory. Therefore, it is challenging to perform deep-learning side-channel attacks on resource-constrained devices.**

In this paper, we propose a framework, named TinyPower, which leverages pruning to reduce the number of parameters of neural networks for side-channel attacks. Pruned neural networks obtained from our framework can successfully run side-channel attacks with a much lower number of parameters and significantly less memory. Our framework focuses on structured pruning over filters of Convolutional Neural Networks (CNNs). We demonstrate the effectiveness of structured pruning over power and EM traces of AES-128 running on microcontrollers (AVR XMEGA and ARM STM32) and FPGAs (Xilinx Artix-7). Our experimental results show that we can achieve a reduction rate of 98.8% (reducing the number of parameters from 53.1 millions to 0.59 millions) on a CNN and still recover keys on XMEGA. For STM32 and Artix-7, we achieve a reduction rate of 92.9% and 87.3% on a CNN respectively. We also demonstrate that our pruned CNNs can effectively perform the attack phase of side-channel attacks on a Raspberry Pi 4 with less than 2.5 millisecond inference time per trace and less than 41 MB memory usage per CNN.

I. Introduction

Deep-learning side-channel attacks [1], [2], [3], [4] utilize neural networks to infer encryption keys on a target, such as a microcontroller or a FPGA (Field Programmable Gate Array). Specifically, a neural network infers intermediate results of encryption based on the correlation between power consumption and intermediate results. Once intermediate results are inferred, correct keys can be recovered based on associated plaintexts and the encryption algorithm. Compared to traditional side-channel attacks, such as Template Attack [5], deep-learning side-channel attacks can defeat countermeasures, such as masking and random delays, and require less pre-processing over raw traces [2].

However, existing neural networks utilized in side-channel attacks are often complex. For instance, the Convolutional Neural Network (CNN) used over ASCAD datasets [2] can consist of more than 53.1 million parameters given 1,000 power samples/measurements per trace (see details in Sec. VI). When performing the attack phase with a trained neural network, a significant amount of memory is needed. This is often not an issue when extensive computation resources (e.g., GPUs) are available. However, it is challenging to run these complex neural networks and perform side-channel attacks on resource-constrained devices.

In this paper, we propose a framework, named *TinyPower*, which significantly reduces the number of parameters of neural networks for side-channel attacks by leveraging the idea of *structured pruning* [6], [7], [8]. Specifically, we focus on structured pruning over *filters*, where less important filters are pruned given a trained CNN. We investigate both *pre-defined* structured pruning and *automatic* structured pruning [7]. Pre-defined structured pruning selects a unique pruning rate across all the layers of a neural network while automatic structured pruning automatically decides a customized pruning rate for each layer. We examine two score algorithms, including l_2-norm [9] and FPGM (Filter Pruning via Geometric Medium) [10], to measure the importance of filters. Our efforts and findings are summarized below:

- We propose a new automatic structured pruning, named MiniDrop, which automatically identifies more important filters in a CNN based on the minimal absolute value of discrete derivative.
- We conduct comprehensive evaluations on public datasets and also datasets collected by us. Specifically, we collect over 2.6 million power and EM traces of unmasked AES-128 from multiple targets, including microcontrollers (AVR XMEGA and ARM STM32) and FPGAs (Xilinx Artix-7) using ChipWhisperer. We utilize EM traces of masked AES-128 on AVR ATMEGA microcontrollers from the well-known ASCAD dataset [2].
- We leverage a CNN used over the well-known ASCAD dataset [2] as a baseline (i.e., a CNN without pruning) for comparison. Our experimental results show that we can achieve a reduction rate of 98.8% (reducing the number of parameters from 53.1 millions to 0.59 millions) on the CNN and still recover keys of AES-128 running on XMEGA. For STM32 and Artix-7, we achieve a reduction rate of 92.9% and 87.3% respectively and remain recovering keys. Structured pruning is also effective when traces are desynchronized, EM, or from masked AES-128.
- Between pre-defined structured pruning and automatic structured pruning, we find that automatic structured pruning achieves a higher reduction rate in the context of side-channel attacks. Between the two score algorithms, we observe that FPGM performs slightly better than l_2-norm in the majority of our experiments.
- We demonstrate that the pruned CNNs we obtain can ef-

fectively perform the attack phase of side-channel attacks on Raspberry Pi 4 Model B (4 GB memory). Specifically, our pruned CNNs are 10X faster in inference time and over 10X lower in memory usage than baseline CNNs.

- Compared to a recent work [11], which reduces the number of parameters of a CNN based on network architecture search with reinforcement learning, our pruned CNNs still have a greater number of parameters. On the other hand, our pruned CNNs require less time to search/prune (e.g., 45 minutes v.s., 3 days) and can still recover keys in the cross-device setting.
- Compared to a recent study (denoted as PWP22) [12] based on unstructured pruning, our pruned CNNs can achieve up to 92.1% reduction rate and still recover keys over EM traces from XMEGA while PWP22 can only achieve up to 1.0% reduction rate over EM traces. Moreover, our method can significantly save memory usage while PWP22 does not.

Reproducibility. Our source code and datasets will be made publicly available upon the acceptance of this paper.

II. Background on Side-Channel Attacks

Side-Channel Attacks (SCAs) can be categorized into *profiling* side-channel attacks and *non-profiling* side-channel attacks. We focus on profiling side-channel attacks in this study.

A. System and Threat Model

System Model. The system model of a profiling side-channel attacks includes a training device and a test device. Specifically, this attacker can capture power/EM traces and associated plaintexts (i.e., inputs of the encryption) from both devices. The attacker knows the key on the training device but not the one on the test device. The attack goal is to reveal the *unknown but fixed* key on the test device.

A profiling side-channel attack includes two phases, the *training phase* (or *profiling phase*) and the *test phase* (or *attack phase*). In the profiling phase, this attacker trains a profile with labeled traces from the training device. Labels of traces are intermediate results of an encryption, which are generated based on the known key and plaintexts. In the attack phase, the attacker acquires unlabeled traces from the test device and tries to recover the unknown key on the test device by leveraging the profile. We focus on deep-learning side-channel attacks, where a profile is a (trained) neural network.

The evaluation of a profiling side-channel attack can be conducted in two settings, including the *same-device setting* and *cross-device setting*. In the same-device setting, we assume that a single device (and a single key) is used for training and attack. In the cross-device setting, two different devices (and two different keys) are used for training and test respectively. Cross-device setting considers potential distribution shifts caused by discrepancies in keys, hardware, and data acquisition. Several studies [13], [14], [15], [16], [17], [18], [19] have shown the importance of evaluating deep-learning side-channel attacks in the cross-device setting.

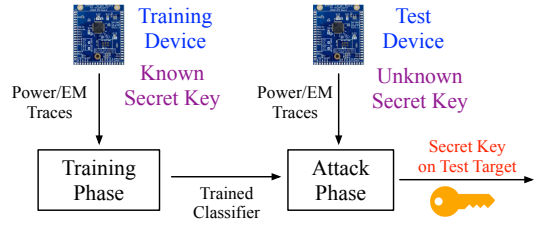


Fig. 1: The system model of profiling side-channel attacks.

For a non-profiling attack, an attacker does not have access to a training device to build a profile in advance. It only has unlabeled traces from the test device to perform the attack.

B. Notations and Leakage Model

A trace t is denoted as a vector $t = (t[1], ..., t[l])$, where $t[i]$ is the measurement of power consumption (or EM radiation) of a device at time i and l is the total number of measurements of a trace. We use $\mathcal{M}$ and $\mathcal{K}$ to denote the plaintext space and the key space respectively. A trace t is acquired when a device runs encryption with plaintext m and key k, where $m \in \mathcal{M}$ and $k \in \mathcal{K}$. We use $z = \varphi(m, k)$ to denote an intermediate value of encryption, where function $\varphi(\cdot)$ is a leakage step.

AES-128. We conduct side-channel attacks on AES-128 encryption (Advanced Encryption Standard), where an encryption key has 16 bytes. The attack performs in a divide-and-conquer approach, where it reveals one key byte each time. Following problem descriptions in the literature [3], [4], we assume key k, plaintext m, or intermediate value z has one byte in the rest of this paper. We use $k_1^*, k_2^*, ..., k_{256}^*$ to denote all the possible 256 key values. We consider the SubBytes of the 1st round of AES as the leakage step $\varphi(\cdot)$.

Leakage Model. We leverage *Hamming Weight (HW) model* or *Identity (ID) model* to formulate side-channel leakage [2], [3]. Both models are able to formulate the side-channel leakage properly. The specific selection between the two leakage models in later evaluations depends on the architecture of a target (microcontroller v.s. FPGA) and the leakage channel (power v.s. EM).

The HW model assumes that there are correlations between the power consumption of an intermediate value and the Hamming weight of this intermediate value. The label of a trace t is $\texttt{HW}(z)$ — the Hamming weight of the intermediate value $z = \varphi(m, k)$ given plaintext m and key k. There are 9 possible Hamming weights (or 9 possible labels) as we assume intermediate value z has one byte. The ID model assumes that there are correlations between the power consumption and the intermediate value itself. The label of a trace is the intermediate value z, and there are 256 possible labels.

Attack Metrics. Given a profile (e.g., a neural network) and a trace, the profile outputs a score for every possible label (either an intermediate value or its HW depending on the leakage model). Then, each score is further assigned to a corresponding possible key based on the label, an associated plaintext, and the encryption algorithm. Every possible key's scores across all the test traces are further aggregated during

the attack. As there are 256 possible keys over a byte, 256 aggregated scores are obtained. Next, the 256 aggregated scores are sorted in a descending order.

We leverage *Key Rank* and *Measurements To Disclosure* (MTD) to measure the effectiveness of side-channel attacks [20], [21]. Key rank r, where $r \in [1, 256]$, is the rank of the aggregated score of the correct key among all the 256 possible keys given a certain number of test traces. A key rank of 1 suggests that the correct key has the highest score and the attacker reveals the key. MTD indicates the number of test traces that is needed for the key rank to converge to 1.

III. Background on Pruning

A. Neural Networks

A neural network consists of a set of layers. A layer consists of a set of filters. A filter is, in essence, an array of weights. A weight $w \in \mathbb{R}$ is a real number indicating the strength of a connection between two neurons in a neural network. In other words, a weight decides how much impact the input of a connection will have on the output of this connection.

Given a neural network consisting of L layers, we use N_{i-1} and N_i to denote the number of input channels and output channels at the i-th layer, where $1 \leq i \leq L$. N_i is also the number of filters at the i-th layer. N_0 is the number of input channels at the first layer. Let $F_{i,j}$ be the j-th filter of the i-th layer, where $F_{i,j} \in \mathbb{R}^{N_{i-1} \times s_i \times s_i}$ and s_i is the kernel size at the i-th layer[1]. The weights of the neural network at the i-th layer can be denoted as $\mathbf{W}^i$, where $\mathbf{W}^i = \{F_{i,1}, ..., F_{i,N_i}\}$. The weights of the entire network can be described as $\mathcal{W} = \{\mathbf{W}^1, ..., \mathbf{W}^L\}$.

B. Neural Network Pruning

Neural Network Pruning [7], [8], or *pruning* for short, is a way of reducing the size of a neural network by removing less important parameters/weights. Pruning can be categorized into two types, *unstructured pruning* and *structured pruning*. Unstructured pruning (i.e., *weight pruning*) removes individual weights. Structured pruning removes parameters in groups (e.g., channels or filters), which is more effective than unstructured pruning. *We focus on structure pruning over filters in Convolutional Neural Networks in this study*.

Structured pruning removes filters that are less important and keeps filters that are more critical to the predictions of a neural network. A *score algorithm* is utilized to measure the importance of a filter. *Pruning rate* p of a layer is defined as the ratio between the number of filters that are pruned/removed and the total number of filters (before pruning) at this layer.

Specifically, given a set of filters $\mathbf{W}^i = \{F_{i,1}, ..., F_{i,N_i}\}$ at the i-th layer and an integer M_i, a pruning algorithm keeps a subset $\mathbf{W}^{i*}$ of M_i filters from all the N_i filters such that the M_i filters have the maximum sum of importance. In other words, the pruning algorithm calculates

$$\underset{\mathbf{W}^{i*} \subset \mathbf{W}^i}{\arg\max} = \sum_{F_{i,j} \in \mathbf{W}^{i*}} S(F_{i,j}) \tag{1}$$

where $|\mathbf{W}^{i*}| = M_i$ and $S(\cdot)$ is a score algorithm. The pruning rate of this layer is defined as $1 - \frac{M_i}{N_i}$. Filters that are in set $\mathbf{W}^i$ but not in subset $\mathbf{W}^{i*}$ are removed.

Structured pruning can be further divided into *pre-defined* structured pruning and *automatic* structured pruning [7], [8]. Pre-defined structured pruning defines a unique pruning rate across all the layers in advance. On the other hand, automatic structured pruning decides a customized pruning rate per layer depending on the score distribution of filters at each layer.

Pruning Process. Given a score algorithm, a pruning process normally includes three steps: (1) training a neural network to converge; (2) pruning less important filters based on a score algorithm and a pruning rate (either pre-defined or automatic); (3) fine-tuning a pruned neural network to regain accuracy in predictions.

Score Algorithms. We investigate two existing score algorithms, including l_2-norm [9] and FPGM (Filter Pruning via Geometric medium) [10], to measure the importance of filters of a neural network in both pre-defined structure pruning and automatic structure pruning. l_2-norm measures the importance of a filter independently while FPGM evaluates the importance of a filter by considering other filters at the same layer.

Specifically, given a filter $F_{i,j} \in \mathbb{R}^{N_{i-1} \times s_i \times s_i}$ at the i-th layer, its l_2 norm (or Euclidean norm) is calculated as

$$S(F_{i,j}) = ||F_{i,j}||_2 = \sqrt{\sum_{x=1}^{N_{i-1}} \sum_{y=1}^{s_i} \sum_{z=1}^{s_i} (w_{i,j}[x][y][z])^2} \tag{2}$$

where $w_{i,j}[x][y][z]$ is an element/weight of filter $F_{i,j}$, for $1 \leq x \leq N_i$, $1 \leq y \leq s_i$, and $1 \leq z \leq s_i$. A higher l_2 norm indicates that a filter is more important.

FPGM [10] utilizes the geometric median of filters at a layer to determine the importance of filters. A filter that is closer to the geometric median is considered less important and can be removed. As computing geometric median directly in a great dimensional space is extremely time-consuming, the authors [10] also proposed an approximate version of FPGM, which calculates the sum of the l_2 distances from a filter to other filters of the same layer. Specifically, given a set of filters $\mathbf{W}^i = \{F_{i,1}, ..., F_{i,N_i}\}$ at the i-th layer, the approximate FPGM score of filter $F_{i,j}$ is computed as

$$S(F_{i,j}) = \sum_{F_{i,l} \in \mathbf{W}^i, l \neq j} ||F_{i,j} - F_{i,l}||_2 \tag{3}$$

A higher FPGM score suggests that a filter is more important. We use the approximate version of FPGM in this paper.

Pruning Metrics. We levearge two metrics, including (1) the *number of parameters* and (2) the number of *Floating Point Operations*, to measure the complexity of a neural network. A FLOP (Floating Point Operation) is an addition,

[1]For a 1D convolutional layer, which is normally utilized over time-series data (e.g., power/EM traces for side-channel attacks), a filter can be represented as $F_{i,j} \in \mathbb{R}^{N_{i-1} \times s_i \times 1}$. For a fully-connected layer, a filter can be described as $F_{i,j} \in \mathbb{R}^{N_{i-1} \times 1 \times 1}$.

subtraction, division, multiplication, or any other operation involving a floating point value. *Reduction Rate* is defined as the ratio between the number of parameters removed after the pruning and the number of parameters in the baseline network before pruning. In addition to measuring reduction rate over parameters, we can also report reduction rate over FLOPs.

IV. TinyPower: Tiny Neural Networks for SCA

In this section, we introduce our framework, TinyPower, which reduces the size of neural networks for side-channel attacks by pruning filters. We leverage l_2-norm and FPGM as the underlying score algorithms in both pre-defined structured pruning and automatic structured pruning.

A. TinyPower with Pre-Defined Structured Pruning

The pre-defined structured pruning for SCA is straightforward. Given a dataset of traces from a target, a baseline neural network is trained in order to recover the key. Given this baseline neural network, we first select a score algorithm (either l_2-norm or FPGM) and choose a pre-defined pruning rate p. Then, a pruned neural network is produced accordingly by removing filters with pruning rate p at every layer of the baseline neural network.

B. TinyPower with Automatic Structured Pruning

For automatic structured pruning, the key question is *how can we decide a customized pruning rate for each layer?* In other words, at each layer, how do we decide which filters should be removed given the scores measured by a score algorithm. We answer this question by proposing a new automatic structured pruning, referred to as *MiniDrop*.

Details of MiniDrop. Given a layer of a trained neural network, MiniDrop first measures scores of filters at this layer based on a score algorithm (l_2-norm or FPGM). Next, given these filter scores, MiniDrop sorts the filters in a descending order based on scores, and finds the filter index with the *minimal absolute value of the discrete derivative* over all the sorted filter scores. We denote this filter index as the *mini-drop filter index*. The filters that are after this mini-drop filter index are considered less important and are removed. On the other hand, the filters that are before (and equal to) this mini-drop filter index are kept. An example is illustrated in Fig. 2.

MiniDrop can be mathematically formulated as below. Given a set of N_i filters $\mathbf{W}^i = \{F_{i,1}, ..., F_{i,N_i}\}$ at i-th layer and a score function $S(\cdot)$, MiniDrop sorts the filters based on their scores in a descending order as $\{F'_{i,1}, ..., F'_{i,N_i}\}$, where $S(F'_{i,j}) \geq S(F'_{i,j+1})$, for $1 \leq j \leq N_i - 1$ and $F'_{i,j} \in W^i$ for $1 \leq j \leq N_i$. Next, MiniDrop computes

$$M_i = \underset{j \in [1, N_i - h]}{\arg\min} \left| \frac{S\left(F'_{i,j+h}\right) - S\left(F'_{i,j}\right)}{h} \right|$$

where $h = 1$ by default. M_i is the mini-drop filter index and also the number of filters that will be kept. Filters $\{F'_{i,1}, ..., F'_{i,M_i}\}$ are kept and filters $\{F'_{i,M_i+1}, ..., F'_{i,N_i}\}$ are removed.

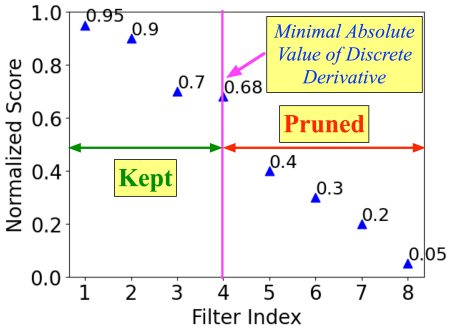


Fig. 2: An example of MiniDrop over 8 filters, where the minimal absolute value of discrete derivative happens at filter index 4 (i.e., $|\frac{0.68-0.7}{1}| = 0.02$). Filters with index 1, 2, 3, and 4 are kept while the rest are pruned.

The *intuition of our idea* is that the minimal absolute value of the discrete derivative reflects the minimal drop between two consecutive filter scores at a given layer. This suggests that the sorted scores drop dramatically *after* the mini-drop filter index, and therefore, filters with much higher scores are included before it. Therefore, the filters before the mini-drop filter index are considered more important and kept. If there are multiple indices minimizing the absolute value of the discrete derivative, the one with the minimal index can be selected.

V. Data Acquisition

Our Power Trace Collection. We examine the results of side-channel attacks on AES-128 over power traces collected from microcontrollers (AVR XMEGA and ARM STM32) and FPGA (Xilinx Artix-7) using ChipWhisperer Level 1 Kits. A ChipWhisperer Level 1 Kit includes a ChipWhisperer-Lite, a CW308 UFO Board, and a target board with a target (either XMEGA or STM32).

Given the same type of targets, we collect data from two different targets for the evaluation of both same-device and cross-device profiling attacks. Specifically, we collect power traces from two XMEGA (X1 and X2), two STM32F3 (S1 and S2), and two FPGAs (F1 and F2). We use the default sampling rate on each target from ChipWhisperer[2]. We always use two different keys between two targets of the same type (e.g., X1 and X2). We leverage TinyAES, which is an unmasked C implementation of AES-128 and can be found in ChipWhisperer repository [22]. The setup of power trace collection is straightforward. ChipWhisperer Lite is powered by a PC through USB. The CW308 UFO Board mounts the target board and is connected to ChipWhisperer Lite.

Our EM Trace Collection. We also collect EM traces from microcontrollers, including AVR XMEGA and ARM STM32. We still use the same C code of unmasked AES-128. In addition to ChipWhisperer, additional hardware are integrated for EM trace collection. The setup is illustrated in Fig. 3.

[2]The sampling rate of data collection on each target: XMEGA: 29.34MHz; STM32: 30.97MHz; FPGA: 96.00MHz. We obtained those by running `scope.clock.adc_freq` command in ChipWhisperer script.

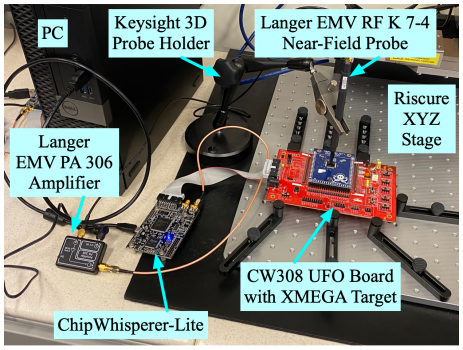


Fig. 3: Our EM trace collection setup

Specifically, we first leverage Riscure XYZ stage to fasten the CW308 UFO Board. Next, we place a Langer EMV RF K 7-4 Near-Field Probe around 2∼5 millimeters above the center of a target and utilize a Keysight 3D Probe Holder to hold the probe. The probe captures EM traces from the target and passes EM traces to ChipWhisperer Lite (and eventually the PC) through a Langer EMV PA 306 Amplifier. We use the same sampling rate for each target as in power trace collection). We set the gain of the scope of ChipWhisperer as 60 (i.e., `scope.gain.gain = 60`).

TinyPower Datasets (Ours). Each dataset we collect is a set of tuples. A tuple is recorded in the form of `(plaintext, trace, key)`. Each trace contains of 5,000 measurements (except 200 measurements per trace for FPGAs). We name each of our datasets by following the format below

`Target_Key_ExtraInfo_NoOfTraces`

where `Target` refers to the target device, `Key` indicates the encryption key, `NoOfTraces` is the number of traces in a dataset, `ExtraInfo` indicates additional information. For instance, a dataset named `X1_K1_EM_150K` indicates that it contains 150,000 EM traces collected from target X1 running key K1. A list of datasets we collected is presented in Table I. The associated keys are reported in Table II.

In addition to the datasets we collected directly, we also generate datasets with random delays. Applying random delays is considered as a countermeasure against traditional side-channel attacks (e.g., Correlation Power Analysis [23]). Specifically, as in previous studies [2], we simulate random delays by delaying measurements of each trace with a random number of r, where $r \in [0, 50]$ and r is uniformly distributed.

Points of Interest. We use *Points of Interest (POI)* to indicate measurements associated with the leakage step in a trace. For example, $POI = [200, 500]$ suggests that measurements from index 200 to index 500 of a trace are associated with the leakage step. For our datasets, we utilize the SubBytes of the first round of AES-128 as the leakage step. We select $POI = [1800, 2800]$ for XMEGA, $POI = [1200, 2200]$ for STM32, and $POI = [0, 100]$ for FPGA. The POIs are identified in advance by manually analyzing the pattern of power traces (with and without the leakage step in C code) and also running statistic analysis, including NICV (Normalized Inner-Class Variance) [24] and SNR (Signal-to-Noise Ratio) [20], when possible. When we train a neural network, we only use measurements within POI of a trace as inputs and measurements outside POI are skipped.

TABLE I: TinyPower Datasets (2.6 million traces, 22.4 GBs)

XMEGA	`X1_K1_200K`	`X2_K2_100K,`
	`X1_K1_Delay_200K`	`X2_K2_Delay_100K`
	`X1_K1_EM_150K`	`X2_K2_EM_150K`
STM32	`S1_K1_200K`	`S2_K3_100K`
	`S1_K1_Delay_200K`	`S2_K3_Delay_100K`
	`S1_K1_EM_150K`	`S2_K3_EM_150K`
FPGA	`F1_K1_200K`	`F2_K4_200K`

TABLE II: Encryption Keys

K1	`0x2b7e 1516 28ae d2a6 abf7 1588 09cf 4f3c`
K2	`0xaa80 d8a7 84d3 3f5c 0b90 a985 208e ff4a`
K3	`0xd2d5 0168 8283 9143 969e e9a2 53a7 52e1`
K4	`0xe6de 35a9 a523 19df c6cc bbba c136 c3bf`

ASCAD Dataset. ASCAD dataset [2] includes 50,000 training traces and 10,000 test traces collected from AVR ATMEGA running masked AES-128, which was implemented in assembly. Each trace contains 100,000 measurements. We use the same $POI = [45400, 46100]$ as in [2], where the 700 measurements from $[45400, 46100]$ are associated with the 3rd byte of the output of SubBytes in 1st round of AES. We only evaluate ASCAD dataset in the same-device setting as there are no cross-device data available.

VI. Evaluation

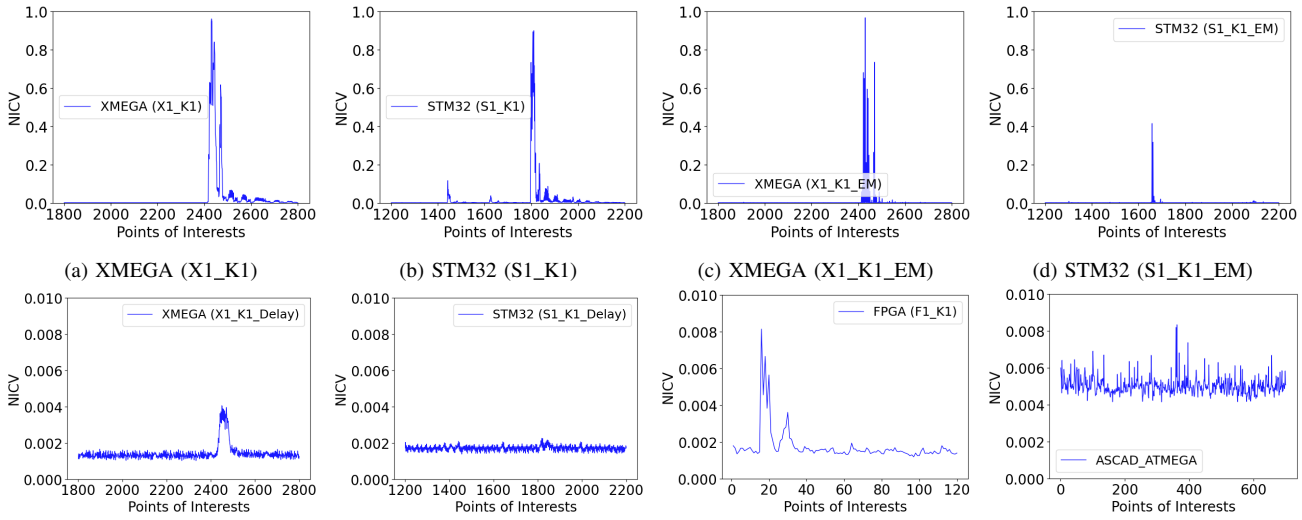
Experiment Setting. We conduct our experiments on a desktop with Ubuntu 18.04, Intel i9 CPU, 64GB memory, and a NVIDIA Titan RTX GPU. We utilize Hamming Weight (HW) model for XMEGA, and Identity (ID) model for STM32, FPGA, and ASCAD to formulate side channel leakage. We always report attack results on the **3rd byte** of an AES key. When we train a CNN, we always use 150 epochs. By default, we use 50,000 traces for training and use 5,000 traces for testing. Depending on the target (e.g., FPGA) or leakage channel (e.g., EM), we also increase the number of training traces when necessary.

For the evaluation of the same-device setting, the training traces and test traces are from the same dataset, which was collected on target X1, S1 or F1. For the cross-device setting, the test traces are selected from a corresponding dataset captured on target X2, S2 or F2. For instance, we choose traces from X1_K1_200k for training and take traces from X2_K2_100k for testing in the cross-device evaluation of the attacks on XMEGA. When we report MTD (Measurements To Disclosure), we always report the average MTD by running tests over the same set of test traces 20 times but randomly shuffling the order of test traces each time. When we compare multiple neural networks attacking the same target, the same randomly-shuffled test traces are used across multiple networks for a fair comparison.

Baseline CNN. We leverage the CNN that was utilized over the ASCAD dataset [2] as the *baseline CNN* in our evaluation.

TABLE III: Attack Performance of Baseline CNNs

Target	Leakage Model	Training Traces	MTD		Parameters (millions)	FLOPs (millions)	Training Time (seconds per epoch)
			Same-Device	Cross-Device			
XMEGA	HW	50k	5	5	53.1	96.54	10.46
XMEGA_Delay	HW	50k	5	6			
XMEGA_EM	HW	50k	15	1,621			
STM32	ID	50k	2	4	54.1	98.57	9.41
STM32_Delay	ID	50k	6	6			
STM32_EM	ID	140k	26	85			32.50
FPGA	ID	180k	428	3,058	24.7	39.85	9.51
ASCAD_ATMEGA	ID	50k	921	NA	43.6	77.59	13.21



(a) XMEGA (X1_K1) (b) STM32 (S1_K1) (c) XMEGA (X1_K1_EM) (d) STM32 (S1_K1_EM)

(e) XMEGA (X1_K1_Delay) (f) STM32 (S1_K1_Delay) (g) FPGA (F1_K1) (h) ASCAD_ATMEGA

Fig. 4: Side-channel leakage of each dataset measured with NICV.

TABLE IV: Hyperparameters of Baseline CNN [2]

Conv 1	filters: 64; kernel size: 11; stride: 2; Relu
Conv 2	filters: 128; kernel size: 11; stride: 2; Relu
Conv 3	filters: 256; kernel size: 11; stride: 2; Relu
Conv 4∼5	filters: 512; kernel size: 11; stride: 2; Relu
AvgPool 1∼5	pooling size: 2; stride: 2
Dense 1∼2	No. of neurons: 4096; Relu
Output	No. of neurons: 9 (HW) or 256 (ID); softmax

This CNN consists of 5 convolutional blocks (including 1 convolutional layer and 1 average pooling layer per block), 2 fully connected layers, and 1 output layer. The hyperparameters of this CNN are listed in Table IV. This CNN has been widely utilized in recent deep-learning side-channel attacks.

Experiment 1: Performance of Baseline CNN. We first report the attack performance of the baseline CNN across difference targets. We investigate both power and EM traces and study both same-device and cross-device scenarios. As we can see in Table III, this baseline CNN can recover keys from all the targets, even when the side-channel leakage is low (e.g., random delays, EM traces, FPGAs, or masked AES). The side-channel leakage of these targets measured based on NICV are reported in Fig. 4. The number of test traces that is needed to recover keys in the cross-device setting, in general, is greater than the one in the same-device setting. This is consistent with previous studies as there are distribution shifts between the training and test data, *especially for EM traces and FPGAs*. On the other hand, the number of parameters involved in this baseline CNN is extremely large (e.g., over 53 millions for microcontrollers and over 24 millions for FPGAs).

Experiment 2: Performance of Pruned CNN (Pre-Defined Structured Pruning). We evaluate the performance of pruned CNNs, where these pruned CNNs are obtained using pre-defined structured pruning. We leverage l_2-norm and FPGM respectively as the underlying score algorithm. Given a trained baseline CNN from Experiment 1, we choose a pruning rate p to remove filters per convolutional layer or neurons per dense layer accordingly. After the pruning, the pruned CNN is fine-tuned with 150 epochs using the same traces used during the training of the baseline CNN. In our experiments, *fine-tuning* means that the parameters of a pruned CNN are initialized based on the ones obtained after the training of a baseline CNN rather than random values.

As presented in Table V, pre-defined structured pruning can effectively reduce the number of parameters in a CNN. For instance, it reduces the number of parameters from 53.1 millions to 2.1 millions (i.e., only 3.96% of the baseline CNN) and still recover keys over power traces on XMEGA. In general, when we increase the pruning rate per layer, the attack performance (i.e., MTD) drops, and eventually fails to recover keys at some point. We have similar observations

TABLE V: Attack Performance of Pruned CNNs, **Pre-Defined Structured Pruning** (MTD = $\perp$ indicates failing to recover keys within 5,000 test traces.)

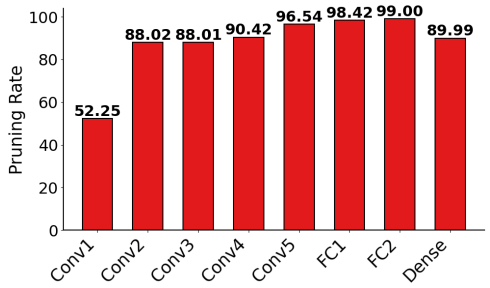
Target	p	Parameters (Reduction Rate)	FLOPs (Reduction Rate)	l_2-norm MTD		FPGM MTD	
				Same	Cross	Same	Cross
XMEGA	0.7	4.77M (90.9%)	8.67M (91.0%)	7	8	6	7
	0.8	**2.12M (96.0%)**	**3.86M (96.0%)**	**95**	**102**	**10**	**10**
	0.9	0.53M (99.0%)	0.97M (98.9%)	$\perp$	$\perp$	$\perp$	$\perp$
XMEGA_Delay	0.7	4.77M (90.9%)	8.67M (91.0%)	7	9	7	9
	0.8	**2.12M (96.0%)**	**3.86M (96.0%)**	**12**	**14**	**12**	**14**
	0.9	0.53M (99.0%)	0.97M (98.9%)	$\perp$	$\perp$	$\perp$	$\perp$
STM32	0.5	13.78M (74.4%)	25.16M (76.0%)	4	3	3	3
	0.6	**8.88M (83.5%)**	**16.20M (83.5%)**	$\perp$	$\perp$	**15**	**15**
	0.7	5.07M (90.6%)	9.28M (90.5%)	$\perp$	$\perp$	$\perp$	$\perp$
STM32_Delay	0.5	13.78M (74.4%)	25.16M (76.0%)	12	10	4	4
	0.6	**8.88M (83.5%)**	**16.20M (83.5%)**	$\perp$	$\perp$	**4**	**4**
	0.7	5.07M (90.6%)	9.28M (90.5%)	$\perp$	$\perp$	$\perp$	$\perp$
XMEGA_EM	0.4	19.09M (64.0%)	34.74M (64.0%)	17	2,459	14	1,570
	0.5	**13.28M (74.9%)**	**24.15M (74.9%)**	**15**	$\perp$	**17**	**3,855**
	0.6	8.47M (84.0%)	15.42M (74.9%)	1,525	$\perp$	23	$\perp$
STM32_EM	0.2	34.72M (35.8%)	63.34M (35.7%)	22	156	21	133
	0.3	**26.69M (50.6%)**	**48.70M (50.6%)**	**22**	**108**	**22**	**167**
	0.4	19.70M (63.6%)	35.96M (63.5%)	27	124	$\perp$	$\perp$
FPGA	0.3	12.32M (50.1%)	19.96M (49.9%)	416	4,337	391	4,439
	0.4	**9.14M (62.9%)**	**14.84M (62.8%)**	**526**	**4,347**	**484**	$\perp$
	0.5	6.44M (73.9%)	10.49M (73.7%)	$\perp$	$\perp$	$\perp$	$\perp$
ASCAD_ATMEGA	0.7	4.13M (90.9%)	7.40M (90.5%)	743	NA	614	NA
	0.8	**1.91M (95.8%)**	**3.43M (95.6%)**	**516**	NA	**578**	NA
	0.9	0.53M (98.8%)	0.96M (98.7%)	$\perp$	NA	$\perp$	NA

over traces from other targets. Specifically, 83.5% (STM32) and 95.8% (ASCAD_ATMEGA) of parameters can be reduced respectively while still being able to reveal keys.

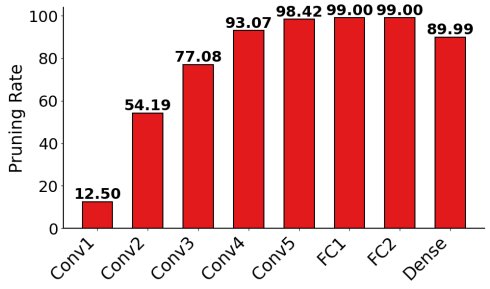
For pruned CNNs over EM traces from microcontrollers or power traces from FPGAs, pruning is also effective but the reduction rate is not as high as others. For example, we achieve a reduction rate of 74.9% (XMEGA_EM), 50.7% (STM32_EM), and 62.9% (FPGA) respectively. This is expected as it is more challenging to recover keys from these targets/cases, and therefore, less parameters can be pruned. For most of the targets, FPGM shows slightly better attack performance than l_2-norm given the same pruning rate.

Experiment 3: Performance of Pruned CNN (Automatic Structured Pruning). We evaluate the performance of pruned CNNs, which are obtained using automatic structured pruning with MiniDrop. We still leverage l_2-norm or FPGM as the score algorithm. Given a trained baseline CNN from Experiment 1, MiniDrop produces a pruned CNN. A maximal pruning rate of 0.99 is set in our experiment to avoid over-pruning at each layer. After the pruning, the pruned CNN is fine-tuned with 150 epochs (same as in Experiment 2).

As presented in Table VI, automatic structured pruning with MiniDrop is effective and can reduce more parameters than pre-defined structured pruning. For instance, automatic structured pruning with MiniDrop can reduce 92.3% of parameters (v.s. 74.9% in pre-defined structure pruning) and still recover keys on XMEGA_EM. In general, when we decrease the value of pruning parameter h in MiniDrop, more parameters can be removed and attack performance drops (and eventually fails to recover keys at some point). We have similar observations from other targets. Moreover, we examine the specific pruning rate



(a) XMEGA (h=8)



(b) ASCAD_ATMEGA (h=8)

Fig. 5: The pruning rate at each layer of a pruned CNN produced by MiniDrop.

at each layer given MiniDrop. As presented in Fig. 5, more filters and nodes can be pruned in later layers of a CNN.

Experiment 4: Comparison between Ours and Previous Studies. We compare structured pruning with two recent methods [12], [11] by using our datasets from XMEGA. Specifically, Perin et al. [12] examine unstructured pruning in the context of side-channel attacks. They utilize unstructured

TABLE VI: Attack Performance of Pruned CNN, **Automatic Structured Pruning** (MTD = $\perp$ indicates failing to recover keys within 5,000 test traces.)

Target	h	l_2 norm				FPGM			
		Parameters (Reduction Rate)	FLOPs (Reduction Rate)	MTD		Parameters (Reduction Rate)	FLOPs (Reduction Rate)	MTD	
				Same	Cross			Same	Cross
XMEGA	8	0.76M (98.6%)	1.21M (98.7%)	7	8	0.96M (98.2%)	1.34M (98.6%)	7	8
	4	0.54M (99.0%)	0.97M (99.0%)	$\perp$	$\perp$	**0.59M (99.0%)**	0.97M (99.0%)	**10**	**11**
	2	0.53M (99.0%)	0.96M (99.0%)	$\perp$	$\perp$	0.53M (99.0%)	0.96M (99.0%)	$\perp$	$\perp$
XMEGA_Delay	4	**0.56M (99.0%)**	0.96M (99.0%)	**14**	**12**	**0.56M (99.0%)**	0.96M (99.0%)	**10**	**12**
	2	0.54M (99.0%)	0.96M (99.0%)	312	285	0.53M (99.0%)	0.97M (99.0%)	1,268	783
	1	0.53M (99.0%)	0.97M (99.0%)	$\perp$	$\perp$	0.53M (99.0%)	0.97M (99.0%)	$\perp$	$\perp$
STM32	16	**4.30M (92.1%)**	5.06M (94.9%)	**4**	**4**	**3.81M (93.0%)**	3.75M (96.2%)	**3**	**5**
	8	1.31M (97.6%)	2.11M (97.9%)	$\perp$	$\perp$	1.28M (97.6%)	1.86M (98.1%)	$\perp$	5
	4	0.69M (98.7%)	1.23M (98.8%)	$\perp$	$\perp$	0.69M (98.7%)	1.23M (98.8%)	$\perp$	$\perp$
STM32_Delay	8	1.39M (97.4%)	1.98M (98.0%)	$\perp$	$\perp$	1.00M (98.2%)	1.42M (98.6%)	5	5
	4	0.68M (98.7%)	1.17M (98.8%)	$\perp$	$\perp$	**0.68M (98.7%)**	1.17M (98.8%)	**4**	**6**
	2	0.63M (98.8%)	1.16M (98.8%)	$\perp$	$\perp$	0.64M (98.8%)	1.17M (98.8%)	$\perp$	$\perp$
XMEGA_EM	32	6.14M (88.4%)	5.80M (94.0%)	14	1,033	4.94M (90.7%)	5.80M (94.0%)	16	4,538
	16	**4.93M (90.7%)**	5.18M (94.6%)	**15**	**3,005**	**4.11M (92.3%)**	5.29M (94.5%)	**19**	**2,913**
	8	1.09M (98.0%)	1.65M (98.3%)	$\perp$	$\perp$	1.33M (97.5%)	1.83M (98.1%)	2,180	$\perp$
STM32_EM	128	15.16M (72.0%)	20.76M (78.9%)	18	149	11.82M (78.1%)	17.99M (81.7%)	23	211
	64	**6.07M (88.8%)**	6.93M (99.0%)	**25**	**189**	**5.78M (89.3%)**	6.33M (93.6%)	**24**	**138**
	32	3.83M (92.9%)	4.11M (95.8%)	$\perp$	$\perp$	3.27M (94.0%)	4.11M (95.8%)	$\perp$	$\perp$
FPGA	32	**3.16M (87.2%)**	0.83M (97.9%)	**415**	**3,826**	**3.13M (87.3%)**	0.84M (97.9%)	**470**	**3,317**
	16	2.48M (90.0%)	0.82M (97.9%)	397	$\perp$	1.64M (93.4%)	0.82M(97.9%)	$\perp$	
	8	1.02M (95.9%)	0.58M (98.5%)	$\perp$	$\perp$	0.66M (97.3%)	0.58M (98.5%)	$\perp$	$\perp$
ASCAD_ATMEGA	8	0.68M (98.4%)	1.04M (98.7%)	872	NA	0.75M (98.3%)	0.96M (98.8%)	3,253	NA
	4	0.58M (98.7%)	0.96M (98.8%)	759	NA	**0.58M (98.7%)**	0.96M (98.8%)	**720**	NA
	2	**0.53M (98.8%)**	0.96M (98.8%)	**1,628**	NA	0.53M (98.8%)	0.96M (98.8%)	$\perp$	NA

TABLE VII: Comparison among the smallest (but still effective) CNNs obtained by different methods. Target: XMEGA

Target	Method	Parameters	Pruning/Search Time (hrs)	CNN Memory Usage (MBs)	MTD	
					Same	Cross
XMEGA	Network Search [11]	519	70.67	0.45	18	$\perp$
	Unstructured Pruning [12]	534,600 (99.0% pruned)	1.30	414.05	145	124
	Structured Pruning (Ours)	587,628 (98.9% pruned)	0.75	5.64	10	11
XMEGA_EM	Network Search [11]	1,077	68.29	3.45	50	$\perp$
	Unstructured Pruning [12]	52.53M (1.0% pruned)	1.30	424.08	45	$\perp$
	Structured Pruning (Ours)	4.11M (92.3% pruned)	0.50	36.30	19	2,913

TABLE VIII: Hyperparameters Search Space for Network Architecture Search with Reinforcement Learning [11]

Maximum Total Layers	14
Maximum Fully Connected Layers	3
Fully Connected Layer Size	[2, 4, 8, 16, 32, 64, 128, 256]
Convolutional Padding Type	SAME
Convolutional Layer Depth	[2, 4, 8, 16, 32, 64, 128]
Convolutional Layer Kernel Size	[1, 2, 3, 25, 50, 75, 100]
Convolutional Layer Stride	1
Pooling Layer Filter Size	[2, 4, 7, 25, 50, 75, 100]
Pooling Layer Stride	[2, 4, 7, 25, 50, 75, 100]
SoftMax Initializer	Glorot Uniform
Initializer for other layers	He Uniform
Activation function	SeLU

pruning (with Lottery Ticket Hypothesis [25]) to reduce the number of parameters of CNNs. As no public source code is provided, we implement the unstructured pruning by following the descriptions in [12].

In addition to pruning, network architecture search is also a way of reducing the size of a neural network by searching groups of parameters in an substantial space. Compared to pruning, network architecture search, in general, is extremely time-consuming. Rijsdijk et al. [11], which is an existing study addressing network architecture search for side-channel attacks, leverage *reinforcement learning* to search the optimal CNN architecture given a dataset. We utilize their source code [11] over our datasets to search for small CNN architectures in order to compare pruning and network architecture search in the context of side-channel attacks.

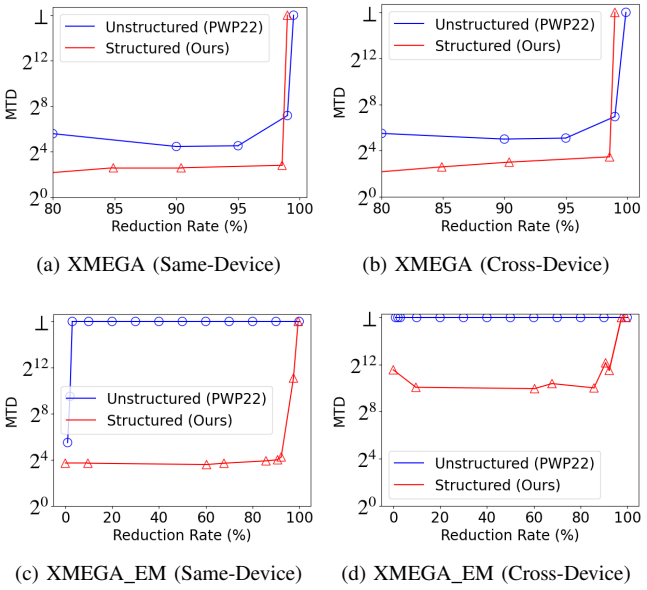
Specifically, given a dataset collected from XMEGA (X1_K1_200k), we tune the hyperparameters of a CNN following the *Reward Small* setting described in [11]. Given the dataset, we select 50,000 traces as training data and 5,000 traces as test data (same as pruning). As the search is extremely time-consuming, we only search over 700 unique CNN models and train each CNN model for 150 epochs. The hyperparameter search space is listed in Table VIII. The search time takes about 70.67 hours in total.

We report the smallest CNN that can still recover keys from each method in our comparison. As shown in Table VII, network architecture search can obviously obtain a much smaller CNN compared to pruning. However, it takes much longer time to search and it does not recover keys in the cross-device setting[3].

[3]We actually tested the top-50 CNN models obtained from network architecture search, none of them recover keys in the cross-device setting.

TABLE IX: Comparison between Baseline CNN and Pruned CNN on Raspberry Pi 4 (Target: XMEGA, STM32)

Dataset	Pruning Option	Parameters	CNN Size (MBs)	Attack Time Per Trace (ms)	CNN Memory Usage (MBs)	MTD (Cross-Device)
XMEGA	No Pruning	27.89M	212.9	10.6	356.7	4 (Setup 1)
	Automatic (h=4)	0.3M	2.4	1.37	23.3	7 (Setup 1) 20 (Setup 2)
STM32	No Pruning	28.90M	220.6	10.5	367.7	3 (Setup 1)
	Automatic (h=16)	1.3M	10.7	2.28	40.2	9 (Setup 1) 22 (Setup 2)



(a) XMEGA (Same-Device) (b) XMEGA (Cross-Device)

(c) XMEGA_EM (Same-Device) (d) XMEGA_EM (Cross-Device)

Fig. 6: Comparison between unstructured pruning (PWP22 [12]) and structured pruning (ours). MTD $= \perp$ indicates failing to recover keys within 5,000 test traces.

Compared to unstructured pruning in [12] (denoted as PWP22), our method can achieve a much higher reduction rate when recovering keys over EM traces as shown in Table VII. A more comprehensive comparison between PWP22 and ours is illustrated in Fig. VII, where we report the attack performance of pruned CNNs across different pruning parameters given each method. We can observe that both PWP22 and our method can significantly reduce the size of a CNN over power traces while still being able to recover keys. Our method can still achieve up to 92.3% reduction rate (for both same-device and cross-device) over EM traces. Unfortunately, PWP22 can only achieve 1.0% reduction rate in the same-device setting and is unable to reduce the size of a CNN in cross-device setting over EM traces. It is also worth mentioning that PWP22 (or unstructured pruning in general) does not save memory usage as removed parameters are set as zeros but are still carried in a neural network.

Experiment 6: Performance of Pruned CNN on Raspberry Pi. We show that a pruned CNN obtained from our method can be effectively operated on a resource-constrained device for the attack phase of side-channel attacks. Specifically, we use a Raspberry Pi 4 Model B running Debian Version 11 (64-bit) with Broadcom BCM2711, quad core Cortex-A72

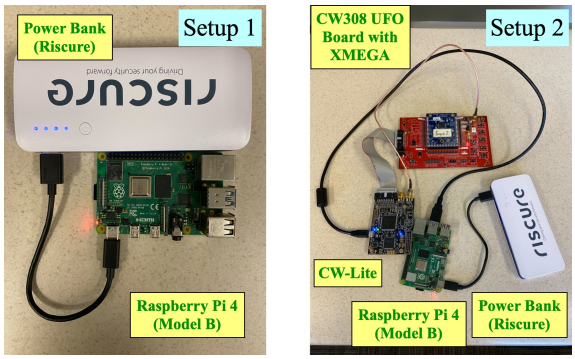


Fig. 7: Our two setups for running profiling side-channel attacks on Raspberry Pi 4. Setup 1 runs the attack phase over pre-collected test traces. Setup 2 runs the attack phase over on-the-fly test traces with ChipWhisperer.

(ARM v8) 64-bit SoC@1.8GHz, and 4GB LPDDR4-3200 SDRAM. We first train and prune a CNN on our GPU machine (same as in our previous experiments). Then, we load the pruned CNN and pre-collected test traces on the Raspberry Pi to perform the attack phase (by using Setup 1 in Fig. 7). We measure the number of parameters, MTDs (cross-device only), network size, network memory usage, and inference/attack time per trace. We examine power traces from XMEGA and STM32. In order to deploy a baseline neural network on Raspberry Pi for comparison, we reduce the POIs (POI=[2400, 2600] for XMEGA and POI=[1700, 1900] for STM32) during the training and attack for baseline CNNs. POIs of Pruned CNNs are also adjusted accordingly for a fair comparison. We only examine automatic structured pruning with FPGM as the score algorithm.

As shown in Table IX, both baseline and pruned CNNs are able to run on Raspberry Pi and recover keys. However, the baseline CNNs are much greater in terms of memory usage and perform much slower. For instance, the baseline CNN over power traces from XMEGA requires 356.7 MBs memory while our pruned CNN only needs 23.3 MBs memory[4]. In addition, the inference time per trace is only 1.37 milliseconds with our pruned CNN while the baseline CNN needs over 10 milliseconds. We have consistent observations from STM32.

Besides loading pre-collected test traces with Setup 1, we also capture test traces on-the-fly and perform the attacks with

[4]It is worth mentioning that we also use Raspberry Pi 3 (1 GB memory) rather than Pi 4 Model B in our evaluation. We can still run a pruned CNN on Raspberry Pi 3 but not a baseline CNN due to limited memory. It is because running the OS alone requires about 400∼500MB memory on Raspberry Pi.

Setup 2 shown in Fig. 7, where a pruned CNN is loaded on Raspberry Pi in advance but ChipWhisperer is connected to Pi to acquire test traces. While test traces are being collected, they are directly passed to a pruned CNN to reveal keys. Our results show that a pruned CNN can still recover keys within 20 traces despite different acquisition setups between training and test power traces.

VII. Related Work

Maghrebi et al. [1] first leverage CNNs for side-channel attacks and show that CNNs outperform Template Attacks. Cagli et al. [26] demonstrate that CNNs can recover keys over desynchronized traces. Benadjila et al. [2] introduce the ASCAD dataset and show CNNs and MLPs (Multi-Layer Perceptrons) can reveal keys over traces from masked AES. Subsequent studies investigate deep-learning side-channel attacks by focusing on different aspects, including portability [13], [14], [15], [16], [27], [17], [18], [28], explainability [29], [30], lack of training (or test) traces [31], loss function [32], training time [33], selection of POIs [34], [35], leakage model [36], or imbalanced data [37]. Recent studies [38], [39], [40], [41] also show that it is feasible to perform non-profiling attacks with deep neural networks. Two comprehensive surveys on deep-learning side-channel attacks can be found in [3], [4]. In addition to applying deep learning to *post-silicon* side-channel attacks mentioned above, some recent research [21], [42] leverage deep learning for *pre-silicon* side-channel attacks over simulated power/EM traces in order to mitigate side-channel leakage during the design stage of chips.

Pruning for SCA. Perin et al. [12] are the first ones explore pruning in the context of side-channel attacks. They utilize weight pruning (also referred to as unstructured pruning) to reduce the number of parameters of CNNs and apply the Lottery Ticket Hypothesis [25], which uses re-initialize weights rather than randomized weights or trained weights before retraining a pruned CNN. Compared to unstructured pruning, structure pruning (explored in this paper) is generally more efficient in terms of reducing the number of parameters and saving memory usage for inference. Detailed surveys summarizing the latest state on pruning can be found in [7], [8].

Network Architecture Search for SCA. Several studies [43], [44], [44], [45], [11], [12] examine automatic hyperparameter tuning (or network architecture search), which also leads to efficient and small neural networks for side-channel attacks. For instance, Zaid et al. [43] utilize visualization methods, including weight visualization, gradient visualization, and heat maps, to improve feature selection and hyperparameter selection, and therefore, reduce the complexity of CNNs for side-channel attacks. Wouters et al. [44] further improve hyperparameter selection for CNNs based on the methods in [43]. Wu et al. [11] leverage reinforcement learning to search hyperparameters. Wu et al. [45] use Bayesian optimization to tune hyperparameters.

Pruning v.s. Network Architecture Search. Compared to pruning, network architecture search can also lead to smaller neural networks but the searching/tuning process is much longer. It is because the searching/tuning needs to be carried out in a substantial search space in order to find efficient combinations of hyperparameters. In addition, neural networks obtained from network architecture search are often extremely customized for training datasets, and may not work well for cross-device settings where distribution shifts exist between training data and test data [46].

VIII. Limitations and Discussions

Other Model Compression Methods. Pruning can be considered as one of the model compression techniques in deep learning [47], [4]. In addition to pruning, there are several other major model compression approaches, including quantization [48], knowledge distillation [49], and low-rank factorization [50]. These remaining compression techniques (either individually or with combinations of multiple approaches) have not been well-investigated in the context of side-channel attacks, and could lead to much smaller neural networks.

Our Pruning is Not Optimal. Even within the scope of pruning, the pruned CNNs we obtained are obviously not optimal. There are still research that can be done to further improve the reduction rate. For instance, pre-processing over traces, such as selecting a smaller number of POIs, can also minimize the dimension of the inputs, and therefore, reduce the size of a neural network.

Examining One Baseline CNN Only. We only examine one CNN architecture as the baseline. While we believe our pruning methods can also been applied to other CNNs and MLPs, the specific reduction rates derived from other architectures can be different from the ones we report in this paper.

Cross-Device Evaluation is Critical. When reducing the size of neural networks for side-channel attacks, we would like to emphasize the importance of carrying out cross-device evaluations. Otherwise, a pruned neural network is extremely customized for one dataset, but will fail to recover keys from another dataset from the same type of target, especially for EM traces and challenging targets (e.g. FPGA).

Impacts of Tiny Neural Networks for SCA. From the perspective of a malicious attacker, being able to perform the attack phase of deep-learning side-channel attack on a resource-limited device can make the attack stealthy and mobile. From the perspective of a security analyzer, understanding that tiny neural networks are feasible for side-channel attacks is also critical for system designs and countermeasure implementations.

IX. Conclusion

We investigate structured pruning in the context of deep-learning side-channel attacks. With extensive experiments over power and EM traces from multiple targets running unmasked and masked AES-128, we demonstrate that much smaller neural networks can be utilized in the attack phase of side-channel attacks. We also show that the pruned CNNs we obtained can be deployed on a Raspberry Pi.

REFERENCES

[1] 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.

[2] 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, 06 2020.

[3] 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.

[4] 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.

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

[6] Y. LeCun, J. Denker, and S. Solla, "Optimal Brain Damage," *Advances in Neural Information Processing Systems*, 1990.

[7] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the State of Neural Network Pruning?" in *Proc. of the 3rd MLSys Conference*, 2020.

[8] 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.

[9] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

[10] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[11] 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*, pp. 677–707, 2021.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] K. Papagiannopoulos, O. Glamocanin, M. Azouaoui, D. Ros, F. Regazzoni, and M. Stojilovic, "The Side-channel Metrics Cheat Sheet," *ACM Computing Surveys*, vol. 55, no. 10, 2023.

[21] 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.

[22] [Online]. Available: https://github.com/newaetech/chipwhisperer

[23] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *Proc. of CHES'04*, 2004.

[24] S. Bhasin, J. Danger, S. Guilley, and Z. Najm, "NICV: Normalized inter-class variance for detection of side-channel leakage," in *2014 International Symposium on Electromagnetic Compatibility*, 2014.

[25] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

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

[27] 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.

[28] M. Krcek and G. Perin, "Autoencoder-enabled Model Portability for Reducing Hyperparameter Tuning Efforts in Side-channel Analysis," https://eprint.iacr.org/2023/019.pdf.

[29] 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.

[30] T. Yap, A. Benamira, S. Bhasin, and T. Peyrin, "Peek into the Black-Box: Interpretable Neural Network using SAT Equations in Side-Channel Analysis," https://eprint.iacr.org/2022/1247.

[31] C. Wang, J. Dani, X. Li, X. Jia, and B. Wang, "Adaptive Fingerprinting: Website Fingerprinting over Few Encrypted Traffic," in *Proc. of ACM CODASPY'21*, 2021.

[32] G. Zaid, L. Bossuet, F. Dassanace, A. Habrard, and A. Venelli, "Ranking Loss: Maximizing the Success Rate in Deep Learning Side-Channel Analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 1, pp. 25–55, 2021.

[33] L. Wu, G. Perin, and S. Picek, "The Best of Two Worlds: Deep Learning-assisted Template Attack," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 3, pp. 413–437, 2022.

[34] 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.

[35] G. Perin, L. Wu, and S. Picek, "Exploring Feature Selection Scenarios for Deep Learning-based Side-channel Analysis," *TCHES'22*, 2022.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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*, 2022.

[46] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[47] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," https://arxiv.org/pdf/1710.09282.pdf.
[48] Y. Guo, "A Survey on Methods and Theories of Quantized Neural Networks," https://arxiv.org/pdf/1808.04752.pdf.
[49] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge Distillation: A Survey," https://arxiv.org/pdf/2006.05525.pdf.
[50] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.