Like an open book ? Read neural network architecture with simple power analysis on 32-bit microcontrollers

Raphaël Joud, Pierre-Alain Moellic, Simon Pontie, Jean-Baptiste Rigaud

▶ To cite this version:

Raphaël Joud, Pierre-Alain Moellic, Simon Pontie, Jean-Baptiste Rigaud. Like an open book ? Read neural network architecture with simple power analysis on 32-bit microcontrollers. CARDIS 2023 - 22nd Smart Card Research and Advanced Application Conference, Nov 2023, Amsterdam, Netherlands. pp.256-276, 10.1007/978-3-031-54409-5_13 . cea-04607993

HAL Id: cea-04607993
https://cea.hal.science/cea-04607993

Submitted on 11 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Like an Open Book? Read Neural Network Architecture with Simple Power Analysis on 32-bit Microcontrollers

Raphaël Joud[1,2], Pierre-Alain Moëllic[1,2], Simon Pontié[1,2], and Jean-Baptiste Rigaud[3]

[1] CEA Tech, Centre CMP, Equipe Commune CEA Tech - Mines Saint-Etienne, F-13541 Gardanne, France
{name.surname}@cea.fr
[2] Univ. Grenoble Alpes, CEA, Leti, F-38000 Grenoble, France
[3] Mines Saint-Etienne, CEA, Leti, Centre CMP, F-13541 Gardanne, France
rigaud@emse.fr

Abstract. Model extraction is a growing concern for the security of AI systems. For deep neural network models, the architecture is the most important information an adversary aims to recover. Being a sequence of repeated computation blocks, neural network models deployed on edge-devices will generate distinctive side-channel leakages. The latter can be exploited to extract critical information when targeted platforms are physically accessible. By combining theoretical knowledge about deep learning practices and analysis of a widespread implementation library (ARM CMSIS-NN), our purpose is to answer this critical question: *how far can we extract architecture information by simply examining an EM side-channel trace?* For the first time, we propose an extraction methodology for traditional MLP and CNN models running on a high-end 32-bit microcontroller (Cortex-M7) that relies only on simple pattern recognition analysis. Despite few challenging cases, we claim that, contrary to parameters extraction, the complexity of the attack is relatively low and we highlight the urgent need for practicable protections that could fit the strong memory and latency requirements of such platforms.

Keywords: Side-Channel Analysis · Confidentiality · Machine Learning · Neural Network · Model Architecture

1 Introduction

Deployment of Deep Neural Network (DNN) models continues to gain momentum, typically with Internet of Things (IoT) applications with microcontroller-based platforms. However, their security is regularly challenged with works focused on availability, integrity and confidentiality threats [17]. Latter topic keeps gathering growing attention from the adversarial Machine Learning (ML) community with *Model Extraction* [2,7,8] becoming a major concern.

An attacker performs a model extraction attack either to steal a well-trained model performance or to precisely recover its characteristics to obtain a *clone model*. Additionally, it would allow the adversary to enhance his level of control

over the victim system to design more adapted and powerful attacks. Therefore, model extraction attacks have been the subject of a growing number of works in recent years with different adversarial scenarios regarding the level of knowledge related to the model and the training data distribution. In the literature, model architecture and parameters are usually studied separately. Many efforts have been brought to parameters recovery with milestones works relying on the assumption that model architecture is known by the attacker [2,5,7,13,18]. However, such an assumption is quite strong and can be questioned as architecture details are generally not disclosed. However, whatever the attacker's objectives, knowledge of victim model architecture significantly increases adversarial ability.

This work is focused on architecture extraction of models embedded on 32-bit microcontrollers thanks to ARM CMSIS-NN library. Our purpose is to know how far an adversary can extract information from a victim model architecture by jointly exploiting the knowledge of the deployment library and very limited side-channel (EM) traces (averaged trace from a single input). Surprisingly and worryingly, we show that (with methods and classical ML expertise) almost all the most important information and hyper-parameters are reachable because of the high *repetitiveness* of the underlying computations. More particularly, for convolutional neural networks, we demonstrate a *Russian doll* effect: one regular EM pattern is related to one hyper-parameter and zooming in presents other patterns related to others hyper-parameters, and so on.

Illustrations, code and data availability. This work relies on the visual analysis of side-channel traces with many (colored) illustrations to explain our approach. For conciseness purpose, we select the most representative ones. We propose additional contents in a public repository[4] with codes and data (traces).

2 Background

2.1 Neural network models

We consider a supervised deep neural network M_W, with W its internal parameters. When fed with an input $x \in \mathbb{R}^d$, the model outputs a set of *prediction scores* $M_W(x) \in \mathbb{R}^K$ with K, the number of labels. Then, the output predicted label is $\hat{y} = \arg\max(M_W(x))$. We note $\mathcal{A}_M$ the architecture of the model M that corresponds to the organisation of its *layers*. $\mathcal{A}_M$ is defined by the nature of each layer, their connections, size and hyper-parameters (non-trainable ones, e.g. the number of convolutional kernels). We note L the number of layers. In this work, we only consider feedforward models with layers stacked horizontally.

MultiLayer Perceptron (MLP) are composed of several vertically stacked *neurons* (or *perceptron*) called *dense layers* (also named *fully-connected* or even *linear*). Each perceptron first processes a weighted sum of its trainable parameters w and b (called *bias*) with the input $x = (x_0, ..., x_{n-1}) \in \mathbb{R}^n$. Then, it non-linearly maps the output thanks to an *activation function* σ: $a(x) =$

[4] `https://github.com/RJoud/ARCH_XTR.git`

$\sigma(w_0 x_0 + ... + w_{n-1} x_{n-1} + b)$, where a is the perceptron output. For MLP, a neuron from layer l gets inputs from all neurons belonging to previous layer $l-1$.

Convolutional Neural Network (CNN) process input data with a set of convolutional *kernels* (also called *filters*). The kernels are usually low-dimensional squared matrices (e.g. 3×3 for image classification). In addition, kernels third dimension matches the number of channels of the input tensor C_{in} (e.g. for RGB images, $C_{in} = 3$). As such, for a *convolutional layer* (hereafter shorted to conv. layer) composed of K kernels of size $Z \times Z \times C_{in}$ applied to an input tensor of size $H_{in} \times H_{in} \times C_{in}$ (we use square inputs for simplicity), the weight tensor W will have the shape $[K, Z, Z, C_{in}]$ (i.e. $K \cdot C_{in} \cdot Z^2$ parameters without bias, $(K+1) \cdot C_{in} \cdot Z^2$ otherwise). Additionally, The number of output tensor channels is $C_{out} = K$. Eq. 1 is a convolution (without bias) expressed as dot-products between kernel and local regions of the input tensor, classically processed by a sliding window. With Y the output tensor, we have $\forall k, l, n \in [\![0, H_{in}[\![^2 \times [\![0, C_{out}[\![$:

$$Y_{k,l,n} = \sigma\Big(\sum_{m=0}^{C_{in}-1} \sum_{i=0}^{Z-1} \sum_{j=0}^{Z-1} W_{i,j,m,n} \cdot X_{k+i,l+j,m} \Big) \tag{1}$$

Other hyper-parameters are *Padding* P and *Stride* S that, respectively, adds extra dimensions (P) to handle the borders of the input tensor[5] and enables to downsample (by S) the sliding[6]. The output tensor shape is defined as in Eq. 2.

$$H_{out} = \frac{H_{in} - Z + 2 \cdot P}{S} + 1 \tag{2}$$

Usually, a conv. layer is followed by a **pooling layer** that aims at reducing the dimensions of the output tensor by locally reducing it with some statistics. A classical approach is to apply a *Max pooling* or an *Average pooling* with a 2×2 kernel (we note $Z_{pool} = 2$) over the output tensor Y of size $H_{out} \times H_{out} \times C_{out}$ so that the resulting tensor is half the size $(H_{out}/2) \times (H_{out}/2) \times C_{out}$.

Activation functions (σ) inject non-linearity through layers. Typical functions map the output of a dense or conv. layer into specific space like $[0, +\infty[$ (ReLU), $[-1, +1]$ (*Tanh*) or $[0, 1]$ (*Sigmoid*, *Softmax*). A widely used function is the Rectified Linear Unit defined as $ReLU(x) = max(0, x)$. The same activation function is applied to all units of a layer and thus can be functionally considered as an independent layer, as in this work.

2.2 Model deployment on Cortex-M platforms

Several tools are available to deploy DNN on Cortex-M platforms such as TF-LM[7], Cube.MX.AI[8] (STMicroelectronics), NNoM [12] or MCUNet [10]. Most of

[5] With $P = 0$, borders are not considered and the output tensor is shorter
[6] $S = 1$ is the standard default sliding, one element at a time.
[7] `https://www.tensorflow.org/lite/microcontrollers`
[8] `https://www.st.com/en/embedded-software/x-cube-ai.html`

them are based on the CMSIS-NN library from ARM [9]. In this work, we study implementations based on NNoM with CMSIS-NN as back-end.

NNoM [12] (standing for *Neural Network on Microcontroller*) is an open-source, high-level neural network inference library dedicated to microcontrollers. It allows to easily implement DNN models previously trained on Keras-Tensorflow, while supporting complex structures (e.g. Inception or ResNet). Converted NNoM models are layer-wise quantized to reduce memory footprint. Scaling factor of the quantization scheme is restricted to powers of two, allowing to perform efficient shifting operations rather than divisions. Back-end operations can be performed by NNoM's eigenfunctions or by efficient CMSIS-NN ones when compatible.

CMSIS-NN [9] is a collection of optimized neural network basic operations, developed for Cortex-M processor cores. They enhance performance and reduce memory footprint with different optimisation techniques that depend on the target platform. They handle quantized variables on 8 or 16 bits. Implementations considered hereafter manipulate 8-bit variables. Using such variables enables to leverage on Single Instruction Multiple Data (SIMD).

2.3 Model Extraction

The goal of *model extraction* attacks is to steal a *victim model* M_W with different possible adversarial goals [7]. A *task-performance* objective [15,19] is to steal the performance of M_W to reach equal or better one at lower cost (e.g., save prohibitive training time). In that case, knowledge of (exact) victim model architecture or parameter values is not necessary. In a *fidelity* scenario, attacker wants to craft a *substitute model* M'_Θ that mimics M_W as close as possible. M' should provide the same predictions as M (correct and incorrect ones). This *similarity* between M and M' is typically defined by measuring the agreement at the label-level [7], i.e. $\arg\max(M'_\Theta(x)) = \arg\max(M_W(x))$ for every x sampled from a target distribution $\mathcal{X}$. A more complex and optimal objective (*Functionally Equivalent Extraction*) targets equal predictions ($M'(x) = M(x), \forall x \in \mathcal{X}$). Importantly, the strongest *Exact Extraction* attack ($\mathcal{A}_M = \mathcal{A}_{M'}$, $W = \Theta$) is impracticable by simply exploiting input/output pairs from victim model [7]. *Fidelity*-oriented scenarios receive a growing interest because of challenging extraction processes, more essentially for the parameters. When dealing with parameter extraction, a usual assumption is that the adversary knows $\mathcal{A}_M$. This is the case for cryptanalysis-like approaches [2,7], active learning techniques [16,19] and recent efforts relying on physical attacks such as side-channel (SCA) [1,8] or fault injection (FIA) [18,6] analysis. Interestingly, whatever the adversarial goal, victim model architecture is a crucial information: it is compulsory for fidelity scenarios, and its knowledge significantly strengthen attacker's abilities to succeed in task-performance ones [15].

3 Related works and contributions

A growing number of works investigates architecture extraction with physical attacks. Proposed techniques are essentially based on Side-Channel Analysis (SCA) such as Timing Analysis (TA), Simple Power/EM Analysis (SPA/SEMA) or Cache Attacks (CA) targeting various physical targets (FPGA, microcontrollers

Table 1: Related State-of-the-Art works. n.s.: Not Specified

Attack	Physical target	Targeted models	Used technique
[4]	MLaaS	CNN	TA & Regression
[22,21]	FPGA	BNN	SEMA
[11]	FPGA	CNN & ResNet	SEMA
[3]	GPU	CNN	SEMA & TA
[1]	µC	MLP & CNN	CEMA
[20]	µC	CNN	SPA & ML
ours	µC	MLP & CNN	SEMA only

(µC), GPU and cloud services hosting DNN models referenced as MLaaS for Machine Learning as a Service). These works are summed up in Table 1. Usually, architecture and parameters are extracted distinctively, however Batina *et al.* propose in [1] to identify layer boundaries while performing parameters extraction with Correlation EM Analysis (CEMA). Correlation scores are used to deduce if currently targeted neuron belongs to the same layer as previous targeted one or not. In other words, if CEMA *fails*, then that means we pass through the next layer. Since parameters extraction with CEMA is highly challenging, as detailed in [8], this method raises practical issues and, to the best of our knowledge, has not been fully demonstrated. Other SCA techniques have been used like TA as in [4] to recover model numbers of layers. SCA are also occasionally combined with other approaches such as learning-based clone reconstruction [22,18] or ML-based classification of traces among a limited set of architectures [20][9]. More recently, [11] presents a complete analysis of the impact of FPGA implementations of CNN on the related electromagnetic activity and proposes and evaluates an obfuscation-based countermeasure.

In this work, our goal is to highlight leakages related to the architecture of embedded DNN models. We are considering models implemented on a 32-bit microcontroller thanks to NNoM library [12] that relies on the widely used CMSIS-NN module [9]. Both of these tools are open-source and as such, we logically consider the attacker has a total access to their code. Acquisitions presented all along this work are obtained with 16 averaged traces all acquired while performing the inference algorithm from a single input belonging to the test set (not used during model training). We deliberately set in such a minimalist setting to assess the level of information that could be extracted by an adversary that has almost no prior knowledge about the victim model and limited experimental data available. This work is not intended to be exhaustive with regards to the extraction of every hyper-parameter of every possible layer types. As most of reference papers in model extraction field, we focus on most common layers

[9] In [20], authors consider 4 variants of AlexNet, Inceptionv3, ResNet-50 and 101, for a total of 16 architectures.

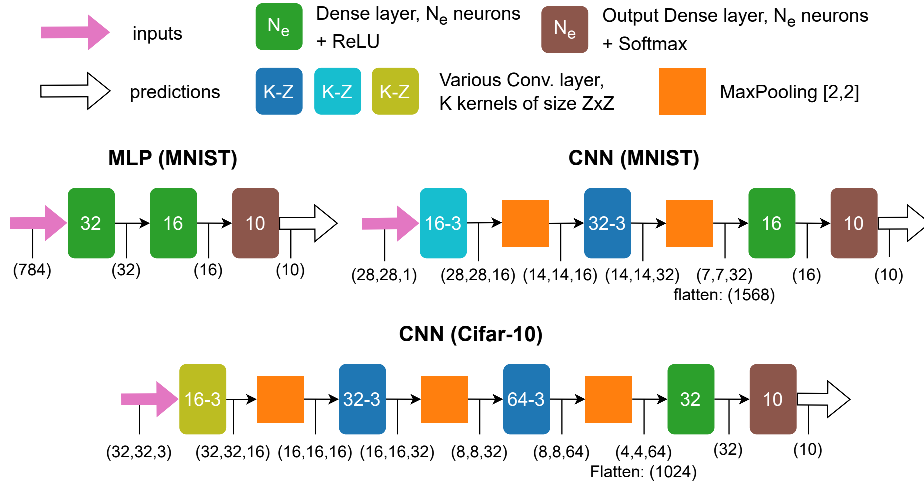


Fig. 1: Detailed architecture of considered models. Various conv. implementations are detailed in Section 6.1

and related parameters, i.e. by targeting MLP and CNN models. Furthermore, all our traces, implementations (and complementary results) are publicly available in order to foster new experiments on this topic that we claim to be highly critical in the actual large-scale model deployment context.

4 Experimental setup

4.1 Models and datasets

We use two traditional benchmarks MNIST[10] (28x28 digit grayscale images, 10 labels) and Cifar-10[11] (32x32 color images, 10 labels). Therefore, we have $H_{in} = 28$ and $C_{in} = 1$ for MNIST and $H_{in} = 32$ and $C_{in} = 3$ for Cifar-10. For MNIST, we trained a MLP and a CNN with the TensorFlow platform that achieved respectively 96% and 98% on the test set. For Cifar-10, we trained a CNN that reached 76% on the test set. Architectures of these models are illustrated in Fig. 1. Colors are used to identify the layer and the shape of each output tensor is shown under each layer. Dense layers (green) have N_e neurons and convolutional layers (blue, olive and cyan) have K square kernels of size Z. Classical stride (= 1) and padding value (*same*) are used, they don't impact the output shape. We only use Max pooling layer (orange) with a kernel size of 2 so that the output tensor is reduced by half. Used activation function is ReLU, except for the last layer with SoftMax function.

[10] `http://yann.lecun.com/exdb/MNIST/`

[11] `https://www.cs.toronto.edu/~kriz/cifar.html`

Table 2: Adversarial objective: list of the hyper-parameters to extract

Target	Parameters	Notation	Target	Parameters	Notation
$\mathcal{A}_M$	# layers	L	Conv. Layer	Output shape	H_{out}
	Type of layers	$\varnothing$		# kernels	K
Dense Layer	# neurons	N_e		Kernel size	Z
MaxPool	Output shape	H_{out}		Stride, Padding	S, P
	Filter size	Z_{pool}	Activation Layer	ReLU or not	$\varnothing$

4.2 Target device and setup

Our experimental platform is an ARM Cortex-M7 based STM32H7 board that can embed state-of-the-art models thanks to its large memories (2 MBytes of Flash and 1 MByte of SRAM). For this first work, interruptions are disabled during model inferences as well as cache optimization available on the board. We measured EM emanations coming from the **unopened** chip with a Langer probe (EMV-Technik LF-U 2,5, freq. range from 100 kHz to 50 MHz) that is connected to an amplifier (Fento HVA-200M-40-F) with a 200 MHz bandwidth and 40 dB gain. Acquisitions are collected thanks to a Lecroy WavePro 604HD-MS oscilloscope. Additional specifications are in the public repository.

5 Threat model

The attacker aims to recover the architecture of an unknown quantized victim model (M_W) with as much detail as possible. Table 2 lists all the information the adversary wants to extract thanks to EM traces. The attack context corresponds to a particular black-box setting. Indeed, the adversary has no knowledge of model architecture nor parameters but is aware of the task performed by the model and the usage of CMSIS-NN module. Adversary is also expected to have appropriate Deep Learning expertise including the most classical and logical layer sequences. The attacker is able to acquire EM side-channel information leaking from the board embedding the targeted DNN model. However, we consider the attacker does not perturb program execution and collect only traces stemming from usual inferences. As we assume a minimal practical setting, adversary is restricted to a single test input. Exploited trace results from the averaging of few inferences with this specific input[12].

Furthermore, we want to point out that in such context, the adversary has access to every needed resources for profiling attacks. Target board is known as well as CMSIS-NN back-end usage, allowing to set a dictionary linking layers with various characteristics to their corresponding EM activity. Simple pattern detection tools could also be used to speedup extraction process. However, usage of such tools or profiling techniques is not in the scope of this work.

[12] Raw traces are available in the public repository.

6 Layers analysis

At the model scale, inference is a feedforward process with the computation of each layer performed one after the other. The output tensor of one layer becomes the input tensor of the next one. In this section, we focus at layer scale with a two-step methodology. First, we analyze the CMSIS-NN implementation of each layer of our models to reveal repetitions of regular computation blocks that should appear in our EM traces. More particularly, we aim to link these *countable* patterns to hyper-parameters of the layer. Second, we experimentally evaluate if these theoretical assumptions are confirmed in our EM traces and assess the complexity of the extraction and potential limitations.

Our approach is based on the principle that the attack is performed according to the computational flow: when the adversary targets layer l, we suppose that analysis of layer $l-1$ is complete and thus its output tensor shape is known, meaning that input tensor shape of layer l is mastered as well[13].

6.1 Convolutional layer

CMSIS-NN convolution functions are all based on the same two-step process, as presented in Alg. 1: Im2col and General Matrix Multiply (GeMM) algorithms. Im2col is a standard optimization trick that implements a convolution through a matrix product rather than several dot-products as defined in Eq. 1. The trick is to prepare all the local areas obtained from sliding window into column vectors and expand the kernel values into rows. Then, convolution is equivalent to a single matrix multiplication that allows an important execution speedup but at the expense of an increased memory footprint. To reduce the latter, CMSIS-NN iteratively performs Im2col with small sets of column vectors [9]. Depending on input and output tensors size, three different functions are proposed[14]:

- `arm_convolve_HWC_q7_fast()` (blue conv. layers in Fig. 1) is for C_{in} multiple of 4 (due to SIMD read and swap) and C_{out} multiple of 2 (due to matrix multiplication applied on 2×2 elements). The computation is speedup for the padding management by splitting the input tensor into 3×3 patches.
- `arm_convolve_HWC_q7_RGB()` (olive conv. layers in Fig. 1) is exclusively optimized for 3 channels inputs (hard-coded condition checks).
- `arm_convolve_HWC_q7_basic()` (cyan conv. layers in Fig. 1) is used otherwise and has a very similar structure as the RGB variant.

Output shape (H_{out})

Code analysis. Whatever the (optimization) differences between the three implementations listed above, extraction of H_{out} relies on the same principle. As presented in Alg. 1, outer loops iterate over the tensor size H_{out} (reminder: we consider square inputs) to run the core computations with the im2col trick then matrix multiplication (GeMM). These computations are time consuming and

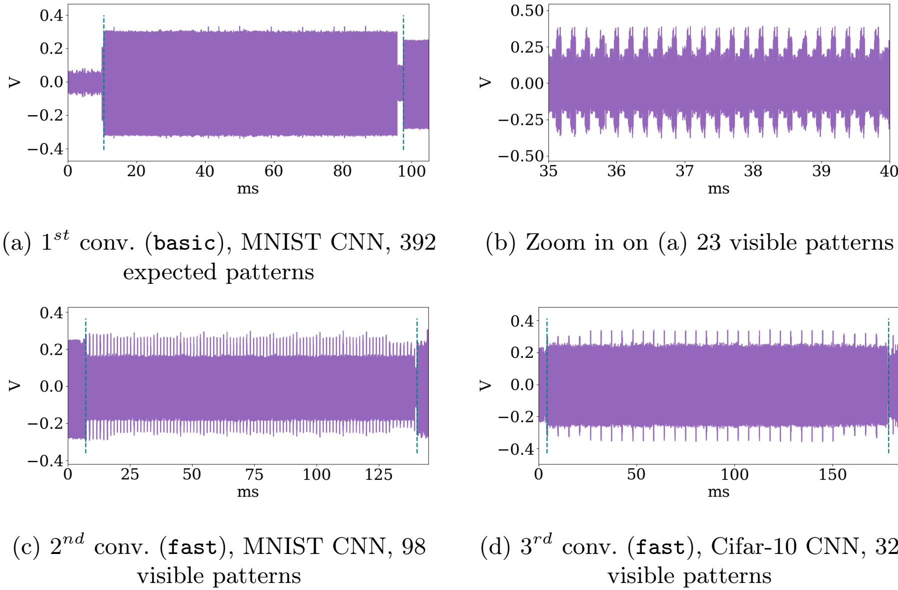
[13] Obviously, we expect that the size of the inputs feeding the model is known by the adversary (e.g., 28×28 for MNIST and $32 \times 32 \times 3$ for Cifar-10).

[14] Equivalent functions are available for non-squared input tensor.

Algorithm 1 General conv. implementation

Input: I_{in} Input tensor of size $H_{in}^2 \cdot C_{in}$, ker (Kernel tensor), S, P, I_{out} Output tensor of size $H_{out}^2 \cdot C_{out}$
Output: Filled I_{out}
1: **for** $i_y \leftarrow 0,\ i_y < H_{out},\ i_y + +1$ **do** ▷ Iterate over H_{out} (y-axis)
2:     **for** $i_x \leftarrow 0,\ i_x < H_{out},\ i_x + +1$ **do** ▷ idem (x-axis)
3:         $\text{buff}_{in} \leftarrow im2col(I_{in}, i_y, i_x, S, P, H_{in}, C_{in})$ ▷ Apply im2col conversion
4:         **if** $len(\text{buff}_{in}) == 2 \times C_{in} \times Z^2$ **then** ▷ Check if 2 input columns are set
5:             $GeMM(\text{buff}_{in}, ker, C_{out}, C_{in} \times Z^2, I_{out})$ ▷ Perform matrix-multiplication
6:             $\text{buff}_{in} \leftarrow 0$ ▷ Buffer reset
7:         **end if**
8:     **end for**
9: **end for**

are likely to induce clear visible EM activities, especially GeMM step (Alg.2 described after). GeMM is called every two iterations (line 4, `if` statement) when the buffer buff_{in} is filled with two *input columns* thanks to im2col (line 3). As a result, GeMM function (line 5) is called $H_{out} \times H_{out}/2$ times during Alg. 1 execution. If we set Np as the number of regular patterns resulting from GeMM function over the part of the EM trace corresponding to the targeted conv. layer, then we can link Np to H_{out} as: $N_p = H_{out} \times H_{out}/2$, i.e. $H_{out} = \sqrt{2 \times N_p}$.



(a) 1st conv. (`basic`), MNIST CNN, 392 expected patterns

(b) Zoom in on (a) 23 visible patterns

(c) 2nd conv. (`fast`), MNIST CNN, 98 visible patterns

(d) 3rd conv. (`fast`), Cifar-10 CNN, 32 visible patterns

Fig. 2: Overviews of conv. layers EM activity related to H_{out}

Observations and limitations. Thanks to this analysis, we assess our traces for each conv. layer of the two CNN models (MNIST and Cifar-10) and observe the correct number of patterns Np as noticed in Tab. 3. Fig 2a illustrates the first conv. layer for MNIST with $Np = 392$ patterns (Fig. 2b gives a zoom on 23 patterns between 35 and 40 ms). Fig 2c and 2d illustrate `fast` conv. layers

of either MNIST or Cifar-10 CNN with respectively $Np = 98$ and 32 patterns. After counting Np, we easily deduce H_{out} values. According to observed layers, number of patterns can be important and hard to count *by hand*. However, since these patterns are clear and accurate (see Fig. 2b and 2d), an attacker could take advantage of basic pattern detection tools (out of the scope of this work) to make the counting easier.

Next, we zoom in one pattern to focus on the GeMM function and look for other regular patterns that may be related to other hyper-parameters, more particularly the number of kernels K and their size Z.

Number of kernels (K)

Code analysis. Matrix-multiplication computation is described in Alg. 2. It mainly consists of an outer `for` loop iterating over the number of output channels (C_{out}) divided by 2, referenced as `rowCnt` and defined at line 1. We remind that for a conv. layer: $C_{out} = K$. Thus, being able to count the iterations of the `for` loop (lines 2-9) enables to recover K. If N_p is the number of regular EM patterns resulting from computations inside this loop, then we assume that $K = 2 \times N_p$.

Algorithm 2 Matrix-product for conv. (**GeMM**)

Input: buff$_{in}$, ker, C_{out}, $C_{in} \times Z^2$, I_{out}, $bias$ (bias tensor)
Output: Partly filled I_{out}

```
1: rowCnt ← C_out >> 1                          ▷ Set rowCnt = K/2
2: for rowCnt > 0, rowCnt − −1 do               ▷ Iterate over K/2
3:     sum, sum1, sum2, sum3 = init_sum(bias, C_in × Z^2)
4:     colCnt ← (C_in × Z^2) >> 2               ▷ Set colCnt as in Eq. 3
5:     for colCnt > 0, colCnt − −1 do
6:         simd_mac(sum, sum1, sum2, sum3, buff_in, ker)
7:     end for
8:     apply_mac(sum, sum1, sum2, sum3, C_out, I_out)
9: end for
10: Manage_remainder_if_any(C_out, buff_in, bias, C_in × Z^2)
```

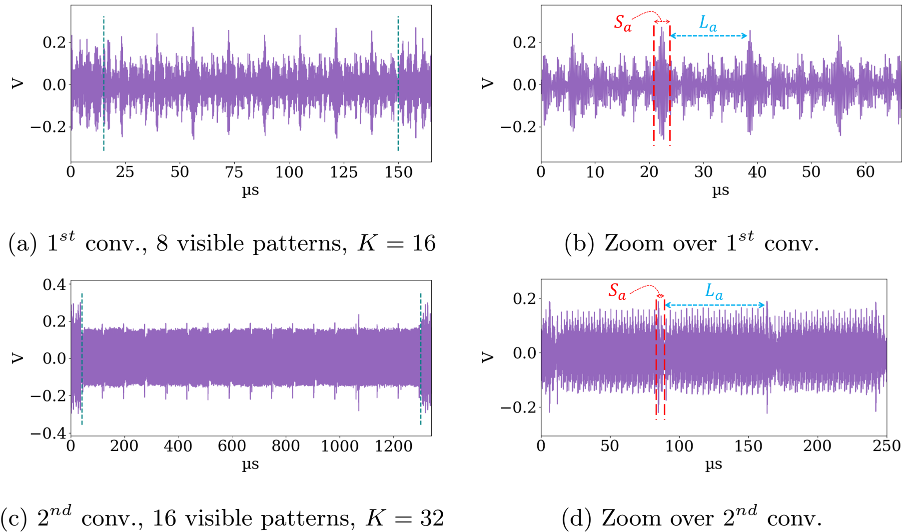
Table 3: H_{out} and expected # patterns

Layer	Mnist H_{out}	Mnist N_p	Cifar-10 H_{out}	Cifar-10 N_p
1	28	392	32	512
2	14	98	16	128
3	∅	∅	8	32

Observations. We evaluate the number of regular patterns appearing on our traces when zooming in the first pattern we previously extracted H_{out} from. For each conv. layer, we observe groups of new regular patterns composed of the expected number N_p. Furthermore, these groups are observed throughout the EM activity related to the entire conv. layer, corresponding to every call of the GeMM function of Alg. 1. These regular patterns are composed of a spike (Sa) followed by a segment of lower frequency activity (La) (Fig. 3b and 3d). Fig 3a and 3c represent targeted outer `for` loop iterations for the first two conv. layers of MNIST CNN with respectively $N_p = 8$ and 16, that corresponds to $K = 16$ and 32 kernels. Such observations have been successfully made for every other conv. layers for both Cifar-10 and MNIST CNN models.

Size of kernel (Z)

Code analysis. Like the number K of kernels, their size Z is also manipulated in Alg. 2. The inner `for` loop (lines 5-7) iterates over the variable `colCnt` which directly depends on Z. This part represents the core computation of convolution



(a) 1st conv., 8 visible patterns, $K = 16$

(b) Zoom over 1st conv.

(c) 2nd conv., 16 visible patterns, $K = 32$

(d) Zoom over 2nd conv.

Fig. 3: Single GeMM execution traces with zoom for MNIST conv. layers

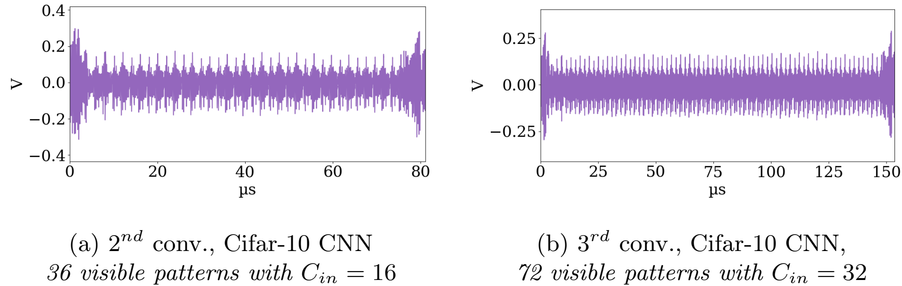
(with SIMD-based accumulations, line 6). `colCnt` assignation (line 4) is as Eq. 3:

$$colCnt = \frac{1}{4}(C_{in} \times Z^2) \tag{3}$$

Thus, as before, we expect that if these very regular computations result in regular EM patterns, then we can estimate Z. However, such a statement is based on the knowledge of C_{in}. As previously mentioned, the attacker is supposed to master the dimensions of the output tensor of the previous layer, meaning that he knows C_{in}. So, if EM activity related to the inner `for` loop can be distinguished from the rest with N_p regular patterns, then Z is estimated as $\sqrt{(4 \times N_p)/C_{in}}$.

Observations. When observing EM activity between two spikes defined previously as La (Fig. 3), repetitive patterns can be spotted as in Fig. 4. EM activities related to SIMD-based accumulations of the second and third conv. layer of Cifar-10 CNN are shown respectively in traces 4a and 4b. For the second layer, $C_{in} = 16$ and $N_p = 36$ patterns can be counted, giving correctly $Z = 3$ according to previous assumption. Same correct result is obtained for the third layer with $C_{in} = 32$ and $N_p = 64$ visible patterns.

Limitations. It is important to note that dividing $C_{in} \times Z^2$ by 4 can induce remainders to be treated separately afterwards. In this case, another EM activity will appear after the `colCnt` expected patterns. Alternatively, patterns to be seen and counted are becoming pretty small and can be hard to distinguish one another. Moreover, the value of C_{in} strongly impacts the number of patterns to differentiate. As an example, number of input channels in the first convolution layer of MNIST CNN is necessarily equal to 1 as input are grayscale images. With 3×3 kernels, only two patterns shall appear, with division remainder



(a) 2nd conv., Cifar-10 CNN
36 visible patterns with $C_{in} = 16$

(b) 3rd conv., Cifar-10 CNN,
72 visible patterns with $C_{in} = 32$

Fig. 4: Zoom in Matrix-product EM activity with kernels of size $Z = 3$

treated afterwards that emits different EM activity. In other words, K can be challenging to recover, especially when C_{in} is small.

Stride (S) and Padding (P)

Once H_{in}, H_{out} and Z are known, based on the fact that $P < Z$ and thanks to Eq. 2, S and P can be deduced. By computing S for possible P values, only a single integer value will result for S.

6.2 Pooling layer: Output dimensions (H_{out}) and kernel size (Z_{pool})

Code analysis. CMSIS-NN implementation (Alg. 3) of max-pooling relies on two computational blocks. First one (lines 1-6) handles the pooling along x-axis with two nested loops iterating over H_{in} and H_{out}. From an adversary's point of view, the second block (lines 8-11) is the most interesting since it completes the pooling over the y-axis with a single loop over H_{out} only. Pooling operation is classically performed with two steps: (1) selection and allocation of local areas, then (2) statistic computation (here, maximum). Thus, we expect to observe two distinct EM activities related to these separated loops with the last (and shorter) one directly related to H_{out}. Then we can deduce $Z_{pool} = H_{out}/H_{in}$.

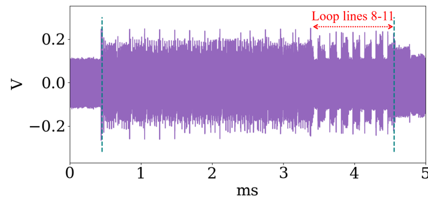
Observations. EM activity of an entire MaxPool layer is represented in Fig 5a (2nd pooling, MNIST CNN). As expected, we observe two distinct blocks of EM emanations, the second one revealing patterns that are more distinguishable. When zooming over this second part, as in Fig 5b (1st pooling, Cifar-10), these patterns are composed of two segments, one with low amplitude activity followed by another of higher amplitude. We guess that it corresponds to the two steps of the pooling computation. The number N_p of these patterns can easily be hand-counted and matches H_{out} value. We have been able to do equivalent observations for all the MaxPool layers of our CNN models. Experimentally, we note that analyzing the first block to retrieve both H_{in} and H_{out} is feasible but more complex than simply counting H_{out} with the second block.

6.3 Dense layer: Number of neurons (N_e)

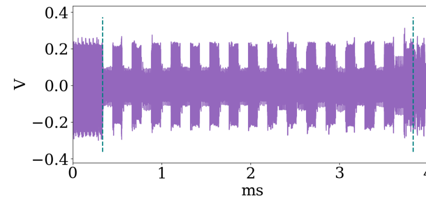
Code analysis. Alg. 4 describes function dedicated to dense layers. It is built around a `for` loop iterating over the `rowCnt` variable that is the number of neurons N_e divided by 4 (line 1), since neurons are handled by groups of 4.

Algorithm 3 MaxPool - `arm_maxpool_q7_HWC`

Input: I_{in} input tenor of size $H_{in}^2 \cdot C_{in}$, I_{out} output tenor of size $H_{out}^2 \cdot C_{out}$, P, S, H_{ker}
Output: Filled I_{out}
1: **for** $i_y \leftarrow 0,\ i_y < H_{in},\ i_y++1$ **do** ▷ Pooling along x axis
2: **for** $i_x \leftarrow 0,\ i_x < H_{out},\ i_x++1$ **do**
3: $win_{start}, win_{stop} \leftarrow set_window(i_y, i_x, I_{in}, H_{ker}, P, S)$
4: $compare_and_remplace_if_larger(win_{start}, win_{stop}, i_y, i_x, I_{in})$
5: **end for**
6: **end for**
7: $trigger_up()$ ▷ Pooling along y axis
8: **for** $i_y \leftarrow 0,\ i_y < H_{out},\ i_y++1$ **do** ▷ Directly iterates over H_{out}
9: $row_{start}, row_{stop} \leftarrow set_rows(i_y, I_{in}, I_{out}, H_{ker}, P, S)$
10: $compare_replace_then_apply(row_{start}, row_{stop}, I_{in}, I_{out})$
11: **end for**
12: $trigger_down()$



(a) Overview of 2nd maxpool of MNIST CNN, $H_{out} = 7$



(b) Zoom over 2nd loop of 1st Maxpool of Cifar-10 CNN, $H_{out} = 16$, triggers set as in Alg. 3 at lines 7 and 12

Fig. 5: MaxPool layers of CNN models

Indeed, `init_sum_with_bias` function (line 3) sets four sum variables. They are used for weighted sum computation (and bias value addition) through multiply-accumulate SIMD operations (`__SMLAD`), performed in `simd_mac` (line 6). We are likely to observe N_p regular patterns emanating from this loop, with $N_e = 4 \times N_p$.

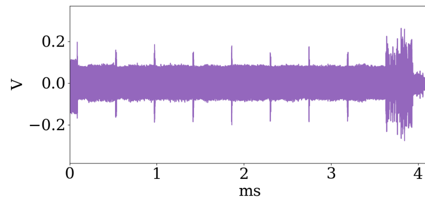
Observations. We observe significant EM activity that results from neurons handling as illustrated in Fig. 6. Observed patterns are mainly composed of uniform blocks separated by clear spikes (especially visible on Fig. 6a). Logically, related EM pattern length directly depends on the number of inputs to the layer and therefore to neurons. Then, dense layers managing broader inputs are easier to analyse. By counting the N_p spike-separated blocks of each dense layers of our MLP and CNN, we checked the link between this number of occurrences N_p and the number of neurons N_e of the layer. Fig. 6a and 6b illustrate the two dense layers of the MNIST MLP model with respectively 8 and 4 patterns corresponding to $N_e = 32$ and $N_e = 16$ neurons. As well, Fig 6c and 6d show dense layers with respectively $N_e = 16$ (4 patterns) and $N_e = 32$ (8 patterns) neurons.

Limitations (special cases). As neurons are grouped in sets of 4, some special cases occur when remaining neurons still need to be computed. The line 10 of Alg. 4 checks remainders and handles them throughout the following `for` loop. Neurons are then handled one by one as only a single sum is initialised then used in `mac` call at line 15. To illustrate this phenomenon, we trained an addi-

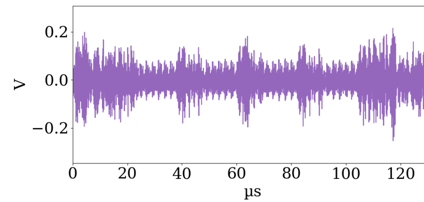
Algorithm 4 Dense layer - `arm_fully_connected_q7_opt`

Input: I_{in} input vector of size H_{in}, ker weight vector of size N_e, $bias$ bias matrix of size N_e, I_{out} output vector of size H_{out}, P, S, H_{ker}
Output: Filled I_{out}

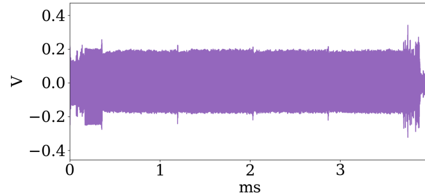
1: $rowCnt \leftarrow N_e >> 2$ ▷ Nb. neurons divided by 4
2: **for** $rowCnt > 0,\ rowCnt - -1$ **do** ▷ Iterate directly over $N_e/4$
3: $sum, sum1, sum2, sum3 = init_sum_with_bias(bias, rowCnt)$
4: $colCnt \leftarrow H_{in} >> 2$
5: **for** $colCnt > 0,\ colCnt - -1$ **do**
6: $simd_mac(sum, sum1, sum2, sum3, ker, colCnt)$
7: **end for**
8: $apply_mac(sum, sum1, sum2, sum3, rowCnt, I_{out})$
9: **end for**
10: $rowCnt \leftarrow N_e\ \&\ 0x3$ ▷ Manage remainders if any
11: **for** $rowCnt > 0,\ rowCnt - -1$ **do**
12: $sum = init_sum_with_bias(bias, rowCnt)$
13: $colCnt \leftarrow H_{in} >> 2$
14: **for** $colCnt > 0,\ colCnt - -1$ **do**
15: $mac(sum, ker, colCnt)$
16: **end for**
17: $apply_mac(sum, rowCnt, I_{out})$
18: **end for**



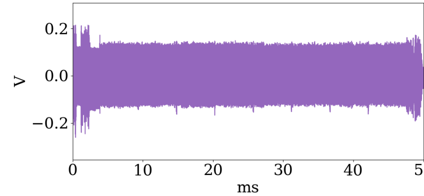
(a) 1st dense layer, MNIST MLP, $N_e = 32$



(b) 2nd dense layer, MNIST MLP, $N_e = 16$



(c) Dense layer, MNIST CNN, $N_e = 16$



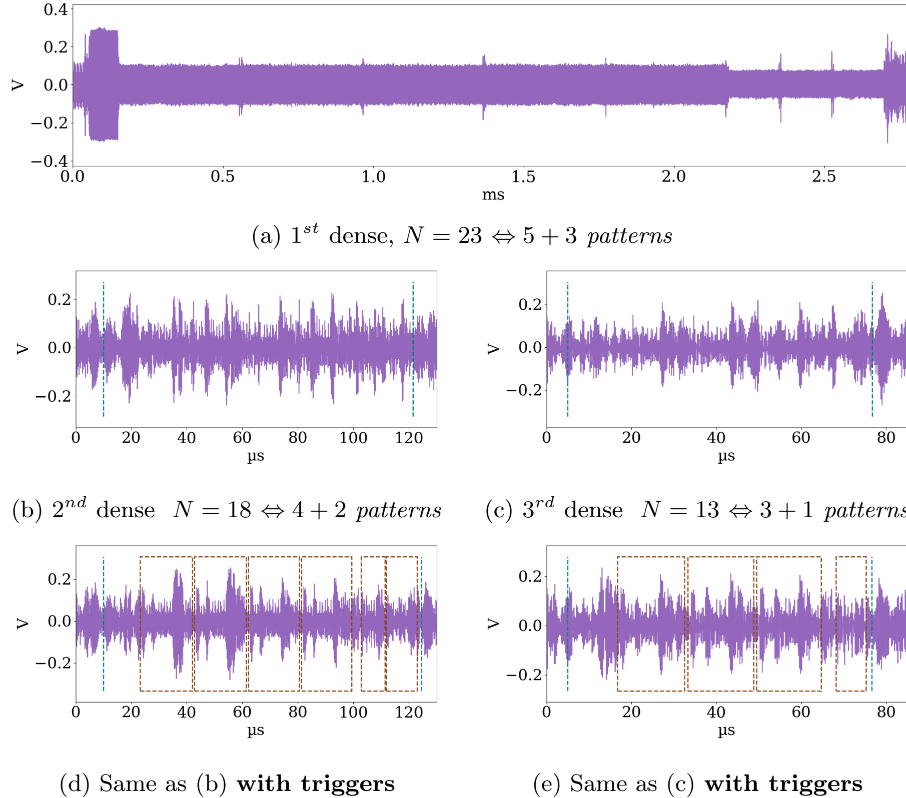
(d) Dense layer, Cifar-10 CNN, $N_e = 32$

Fig. 6: Overview of dense layer EM activity corresponding to general cases

tional MLP model on MNIST (noted SP-MLP), composed of 4 dense layers with respectively 23, 18, 13 and 10 neurons. These correspond to different remainders.

Fig. 7 shows EM activity of each layer for SP-MLP. For the 23-neuron layer, we clearly observe on Fig. 7a 3 blocks that stand out after the core sequence of 5 patterns. This directly matches what is expected with 5 groups of 4 neurons completed with the 3 remaining ones managed independently. However, similar analysis cannot be performed on the two other traces. This difficulty comes from input tensor shape reduction from first layer to the second and third ones. To

verify that neurons are managed in the same way for both of these, triggers are raised and lowered inside the outer `for` loop over `rowCnt` (represented as rectangles in Fig. 7d and 7e). In addition, it illustrates problematic pattern shape changes induced by usage of triggers around observed piece of code. Indeed, they become more visible in Fig. 7d and 7e compared to Fig. 7b and 7c.



(a) 1st dense, $N = 23 \Leftrightarrow 5 + 3$ *patterns*

(b) 2nd dense $N = 18 \Leftrightarrow 4 + 2$ *patterns*

(c) 3rd dense $N = 13 \Leftrightarrow 3 + 1$ *patterns*

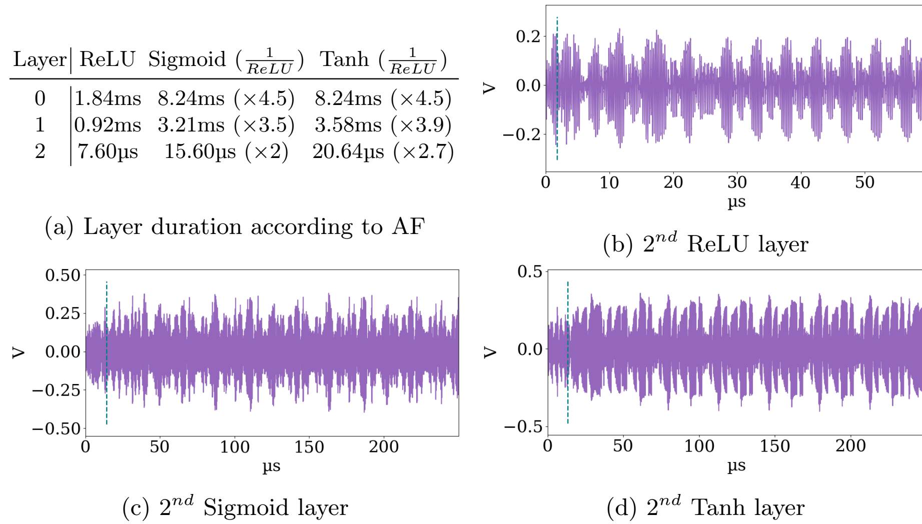
(d) Same as (b) **with triggers**

(e) Same as (c) **with triggers**

Fig. 7: EM activity overview of dense layers for the custom SP-MLP model. Expected patterns clearly appear for the first layer, not for the 2nd and 3rd

6.4 Activation layer

The nature of the Activation Functions (AF) is an important information since it strongly affects input flow from one layer to the next. It is especially useful for parameter extraction as it can modify layers input distribution and potentially their signs (e.g., $\forall x \in \mathbb{R}, ReLU(x) \geq 0$). Many different AF exist with Sigmoid, Tanh, Softmax (mainly for the model output normalization) and ReLU as the most popular. The first three imply an exponential or division computation that are time consuming. ReLU is predominantly the most popular AF because of training efficiency and its low computation requirements. Therefore, we mainly focus on the distinction between ReLU from Sigmoid and Tanh functions (i.e., the extraction goal is to answer the question: *is the AF ReLU or not?*). Mea-



Layer	ReLU	Sigmoid ($\frac{1}{ReLU}$)	Tanh ($\frac{1}{ReLU}$)
0	1.84ms	8.24ms (×4.5)	8.24ms (×4.5)
1	0.92ms	3.21ms (×3.5)	3.58ms (×3.9)
2	7.60µs	15.60µs (×2)	20.64µs (×2.7)

(a) Layer duration according to AF

(b) 2nd ReLU layer

(c) 2nd Sigmoid layer

(d) 2nd Tanh layer

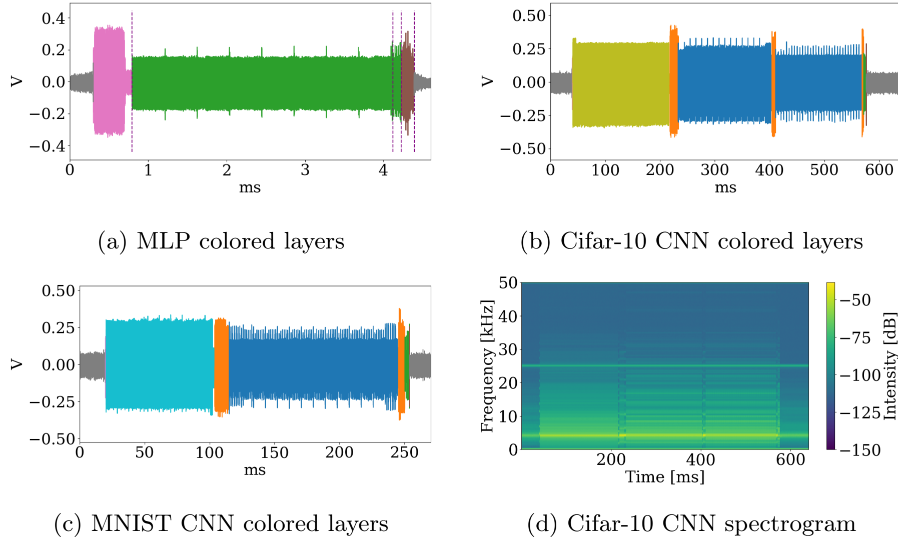
Fig. 8: Zoom over the beginning of second activation layers with their duration

suring computation time of entire AF layer can give strong evidence (as also investigated in [1]) because ReLU function is processed faster than the other two as shown in Table (a) from Fig. 8. Such approach takes on its full meaning when considering potential template capacity of the attacker.

However, we observe that looking at the EM patterns gives additional hints that help distinguish ReLU. For that purpose, we trained three MNIST CNN models with the same architecture but different AF (one per model: ReLU, Sigmoid or Tanh). Traces of Fig. 8 are zoomed in on the beginning of each second AF layer. Trace 8b, corresponding to ReLU, exhibits regular and distinguishable patterns (duration of few μs). Sigmoid and Tanh traces also present groups of peaks repeated throughout traces. However, these are more complex and less explicit than ones related to ReLU and the order of magnitude of their duration is greater, especially for Sigmoid traces. From these observations, distinction between ReLU and the two other AF is quite straightforward. Noteworthy, it is not a clear-cut between Sigmoid and Tanh.

7 Architecture extraction methodology

At first, an adversary may exploit the nature of the task performed by the model. In some cases, it can give hints about the type of the victim model. A model performing a computer vision task is likely to be built around CNN principle. However, there are no strict rules that match a task to a model type. We claim that task knowledge may provide a set of possible layer types that a first analysis of the trace overall structure must confirm. Moreover, preliminary knowledge also encompasses classical deep learning practices, more precisely the *logical* order of layers. If the first layer is identified as a convolutional one, a standard association is another conv. layer or a pooling one.



(a) MLP colored layers

(b) Cifar-10 CNN colored layers

(c) MNIST CNN colored layers

(d) Cifar-10 CNN spectrogram

Fig. 9: Average EM trace for the 3 models. Spectrogram for Cifar-10 CNN.

The first information to extract is the overall structure of $\mathcal{A}_M$ with two information: the number of layers, L and their natures. Then, the attacker focuses on each layer one after the other and, according to their nature, extracts a set of hyper-parameters easing to design a substitute model M'_{Θ}.

Step 1: Finding the number of layers. When analyzing the general shape of the averaged EM trace, the very first observation is that it can be straightforwardly split into several blocks by identifying the boundaries that separate them. A frequency spectrum can highlight the separation between the blocks. The Fig. 9d illustrates how this processing can made easier the layer splitting for the Cifar-10 CNN. Without difficulty, for the three models studied in this work, this simple analysis gives the exact number L of layers for each model architecture $\mathcal{A}_M$. Fig. 9 illustrates the identified blocks for our models with the same colors as in Fig. 1. To validate that layers truly correspond to blocks and for illustration purpose only, triggers have been added to clearly mark out layer boundaries.

Step 2: Identifying layers' nature. Having the position of the L layers on the EM trace, the attacker knows the intrinsic layer complexity that directly influences the execution times. Typically, pooling and AF layers are far less complex than the main computational ones (in our case, dense and conv.) that they usually follow. Thus, the real challenge is to correctly initiate the extraction process by identifying the first layer as a dense or a conv. layer. A correct extraction of the first layer is fundamental, since the *course* of the attack, as well as assumptions made on $\mathcal{A}_M$, come out of this *good start.* To successfully achieve this distinction between dense and conv. layer, the attacker relies on a twofold analysis:

- *Complexity / execution times.* First insight is provided by simple timing analysis. Although insufficient, this analysis may dispel doubts between some

layer hypotheses. Complexity is estimated in different ways in the literature but a standard way is to consider the number of Multiplications and Accumulations (MACs). Such metric directly stems from computations and input/output dimensions. MAC complexity of conv. and dense layers are as follow: $MAC_{Conv2D} = (Z^2 \times C_{in}) \times (H_{out}^2 \times C_{out})$, $MAC_{dense} = H_{in} \times H_{out}$. Complexity of the first layer of both MNIST models are 112896 MACs for the CNN and 25088 MACs (x4.5 less) for the MLP. Thus, usually, conv. layer takes longer to execute than dense one (with similar inputs)[15].
- *Patterns:* the attacker mostly leverages on specific EM activities. As detailed in Section 6, the regularities and repetitions of patterns are very different between dense and conv. layers and confusion is very unlikely. Thus, layer's nature is identified before accurately extracted the related hyper-parameters.

Once the first layer is identified, ML expertise and usual layers sequence knowledge can provide additional hints about next layers nature. This encourages the attacker to extract $\mathcal{A}_M$ by analysing model layers consecutively.

Step 3: Extracting hyper-parameters. Once the overall architecture is known, the attacker follows the analysis presented in Section 6 to recover hyper-parameters.

8 Discussions and Perspectives

Our main objective is to estimate how much information about the architecture of a victim model an adversary can extract by exploiting limited side-channel traces. This information can considerably help the attacker to perform powerful adversarial attacks against the integrity or confidentiality of the victim model. Even though we demonstrate that very critical information can be deduced by the methodical analysis of both an EM trace and the implementation details of the deployment library (here, CMSIS-NN), the use of pattern extraction and recognition tools may significantly ease extraction process by automating most steps and potentially help in extracting more hyper-parameters. We believe that this work paves the way for such further analysis as other efforts that would aim to widen the scope of analyzed architectures and layers. More particularly, batch normalization or tensor arithmetic layers as in state-of-the-art ResNet models or Attention blocks as in Transformer models are relevant candidates for future works. We also highlight that variants of the standard convolution have been proposed for (computational) efficiency purpose (e.g., Depthwise Separable Convolution) [14]. To the best of our knowledge, all these variants rely on highly repetitive and regular computations as the convolution with im2col and GeMM, therefore it should not fundamentally differ from what we exposed in this work.

Moreover, the impact of experimental setup simplifications (i.e., disable of interruptions and cache optimizations) must be studied in complementary studies. Such changes could harden hyper-parameters recovery and directed research focused on protection dedicated to model architecture. To the best of our knowledge, they are very few of them, especially for microcontroller platform. Authors

[15] Obviously, if the dense layer had x4.5 more neurons, MAC complexity between dense and conv. would be equal but having a too large number of neurons (i.e. trainable parameters) for dense layers is usually unsuitable with classical overfitting issues.

of [11] propose an obfuscation strategy for FPGA accelerators. They leverage on optimization parameters of convolution computation to mitigate EM emanations coming from conv. layers. Porting similar protection to microcontroller could be challenging due to limited resources and model performance to be preserved.

9 Conclusion

When dealing with DNN model extraction attack, architecture is crucial. Whatever the attacker's objective, a complete or even partial recovery provides an essential advantage. With this work, we highlight that the attack surface for such a threat is significantly extended by side-channel analysis. More importantly, regarding our application scope, we demonstrate that there is no need for complex exploitation methods (e.g., with heavy supervised profiling step) because of the strong repetitiveness and regularity of most of performed computations that make SEMA a surprisingly powerful tool. Typically, the Russian dolls effect that we exploit for conv. layers (enabling the recovery, one after the other, of several important hyper-parameters) is highly representative of this confidentiality flaw that we claim to be a very worrying concern.

Although we highlight some limitations or more complex special cases that need to be handled in future works, our concern is not based only on the relative simplicity of the attack, but also on the hard challenges related to the development of efficient *and* practical protections, compliant with the constrains of 32-bit microcontrollers. With ongoing regulatory frameworks for AI systems and upcoming security certification actions, model architecture obfuscation appears as the key defense challenge, that we urgently need to solve in order to bring robustness to the large-scale deployment of ML systems.

Acknowledgements

This work is supported by (CEA-Leti) the EU project InSecTT (ECSEL JU 876038)and by ANR (Fr) in the framework *Investissements d'avenir* program (ANR-10-AIRT-05, irtnanoelec); and (Mines Saint-Etienne) by ANR PICTURE program (AAPG2020). This work benefited from the French Jean Zay supercomputer with the AI dynamic access program.

References

1. Batina, L., Jap, D., Bhasin, S., Picek, S.: Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In: 28th USENIX Security Symposium. USENIX Association (2019)
2. Carlini, N., Jagielski, M., Mironov, I.: Cryptanalytic extraction of neural network models. In: Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–21. Springer (2020)
3. Chmielewski, Ł., Weissbart, L.: On reverse engineering neural network implementation on gpu. In: Applied Cryptography and Network Security Workshops: ACNS Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, and SiMLA, Kamakura, Japan. Springer (2021)
4. Duddu, V., Samanta, D., Rao, D.V., Balas, V.E.: Stealing neural networks via timing side channels. arXiv preprint arXiv:1812.11720 (2018)

5. Gongye, C., Fei, Y., Wahl, T.: Reverse-engineering deep neural networks using floating-point timing side-channels. In: 57th ACM/IEEE Design Automation Conference (DAC). IEEE (2020)
6. Hector, K., Moellic, P.A., Dumont, M., Dutertre, J.M.: Fault injection and safe-error attack for extraction of embedded neural network models. In: Workshop on Security and Artificial Intelligence. Springer (2023)
7. Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., Papernot, N.: High accuracy and high fidelity extraction of neural networks. In: Proceedings of the 29th USENIX Conference on Security Symposium (2020)
8. Joud, R., Moëllic, P.A., Pontié, S., Rigaud, J.B.: A practical introduction to side-channel extraction of deep neural network parameters. In: Smart Card Research and Advanced Applications: 21st International Conference, CARDIS 2022, Birmingham, UK, November 7–9, 2022. Springer (2023)
9. Lai, L., Suda, N., Chandra, V.: Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. arXiv preprint arXiv:1801.06601 (2018)
10. Lin, J., Chen, W.M., Lin, Y., Gan, C., Han, S., et al.: Mcunet: Tiny deep learning on iot devices. Advances in Neural Information Processing Systems **33** (2020)
11. Luo, Y., Duan, S., Gongye, C., Fei, Y., Xu, X.: Nnrearch: A tensor program scheduling framework against neural network architecture reverse engineering. In: IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE (2022)
12. Ma, J.: A higher-level Neural Network library on Microcontrollers (NNoM) (2020)
13. Maji, S., Banerjee, U., Chandrakasan, A.P.: Leaky nets: Recovering embedded neural network models and inputs through simple power and timing side-channels—attacks and defenses. IEEE Internet of Things Journal **8**(15) (2021)
14. Nguyen, B., Moëllic, P.A., Blayac, S.: Evaluation of convolution primitives for embedded neural networks on 32-bit microcontrollers. In: International Conference on Intelligent Systems Design and Applications. Springer (2022)
15. Orekondy, T., Schiele, B., Fritz, M.: Knockoff nets: Stealing functionality of black-box models. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (2019)
16. Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z.B., Swami, A.: Practical black-box attacks against machine learning. In: Proceedings of the ACM on Asia conference on computer and communications security (2017)
17. Papernot, N., McDaniel, P., Sinha, A., Wellman, M.P.: Sok: Security and privacy in machine learning. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 399–414. IEEE (2018)
18. Rakin, A.S., Chowdhuryy, M.H.I., Yao, F., Fan, D.: Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In: IEEE Symposium on Security and Privacy (SP). IEEE (2022)
19. Tramèr, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T.: Stealing machine learning models via prediction apis. In: USENIX security symposium. vol. 16 (2016)
20. Xiang, Y., Chen, Z., Chen, Z., et al.: Open dnn box by power side-channel attack. IEEE Transactions on Circuits and Systems II: Express Briefs **67**(11) (2020)
21. Yli-Mäyry, V., Ito, A., Homma, N., Bhasin, S., Jap, D.: Extraction of binarized neural network architecture and secret parameters using side-channel information. In: IEEE International Symposium on Circuits and Systems (ISCAS). IEEE (2021)
22. Yu, H., Ma, H., Yang, K., Zhao, Y., Jin, Y.: Deepem: Deep neural networks model recovery through em side-channel information leakage. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE (2020)