Unsupervised Side-Channel Extraction of Dataflow AI Accelerator Architectures Implemented on FPGA

Mathieu Descos, Brice Colombier, Cédric Killian
Université Jean Monnet Saint-Etienne, CNRS, Institut d Optique Graduate School, Laboratoire Hubert Curien UMR 5516
F-42023, SAINT-ETIENNE, France
{mathieu.descos, b.colombier, cedric.killian}@univ-st-etienne.fr

***Abstract*—FPGAs enable the efficient deployment of neural networks by tailoring the hardware to fit the network. This customization is highlighted in dataflow architectures, where the parallelization of each layer can be chosen. Several frameworks allow a user-friendly implementation by generating the bitfile automatically. However, the widespread use of neural networks, combined with the use of these frameworks that ease their deployment requires to undertake an evaluation of their security. In this work, we propose a side-channel attack based on simple power analysis on a dataflow implementation of neural networks, proving that an attacker can recover the architecture of a neural network model. The unsupervised attack recovers the number of layers of the network and the number of neurons per layer of a Multi Layer Perceptron (MLP) using only the power consumption of the FPGA. This methodology was successfully applied on different MLP architectures trained on the MNIST dataset.**

***Index Terms*—Hardware Security, Neural Networks, Side-Channel Analysis, Hyper-parameters Extraction, Dataflow Accelerator**

I. Introduction

Artificial intelligence is one of the growing topics of the 21st century. This technology is broadly used in everyday objects, including smartphones, computers and autonomous vehicles. Its applications include image recognition and noise cancellation. While the initial usage of AI mainly relied on large models executed on GPUs in data centers, the deployment of neural networks is now also directed towards embedded systems using electronic devices such as microcontrollers and FPGAs. The latter option is of particular interest due to its parallelization of the computation which is adapted to the computation of the neurons and the layers. However the deployment of neural networks on FPGA represents a huge engineering effort. To reduce this effort, frameworks such as FINN [1] and hsl4ml [2] promise to ease the implementation of the networks on FPGA by generating a bitfile from a trained model in Python. The rising deployment of neural networks on FPGA using these frameworks brings the question of the security of the implemented network. Indeed the architecture of neural networks represents valuable intellectual property. Hence the security of these implementations must be studied.

The security threats of neural network implementations has been largely studied on models implemented on microcontrollers. It has been demonstrated that side-channel attacks could recover both the inputs [3] and the architecture of the model [4]. However fewer work has been done on FPGA implementations, and particularly on dataflow architecture implementations. Moreover, in order to reverse engineer a neural network model, an attacker must first focus on the **architecture** of the model, *i.e.* the number of layers, the type of layers and the number of neurons in each layer. Several works assume the attacker has this knowledge already and focus on recovering the weights of the neural network but this represents a strong assumption. Conversely, this work is directed towards recovering the architecture of a neural network running on an FPGA with a dataflow implementation. Existing works on the subject all perform supervised side-channel attacks in two steps: a classifier is first trained on an open device and then used on the target device to recover secret data. This work presents the first unsupervised attack on neural networks implemented on FPGA using a dataflow implementation. The contributions of this work are as follows:

- This paper presents the first reverse engineering attack on a dataflow implementation that does not require profiling.
- We propose a side-channel attack that recovers both the number of hidden layers and the number of neurons per layer by using simple power analysis.
- We successfully applied this methodology to recover the architectures of different MLPs.

This paper is organized in five parts. The necessary background and related works are presented in Section II. Section III presents the attacker model and Section IV details the methodology of the attack. In Section V we explain the experimental setup and discuss the results based on this setup. Section VI concludes this paper.

II. Background and related works

A. Neural networks

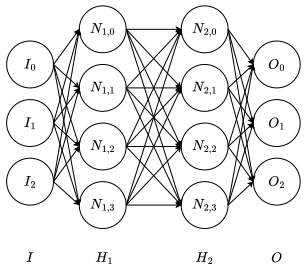


Fig. 1: Architecture of an MLP composed of 2 hidden layers (H_1 and H_2) with 4 neurons each (from $N_{1,0}$ to $N_{2,3}$).

Multi Layer Perceptrons (MLPs) are a type of neural network constructed with an input layer I with n_{input} inputs, n_{layers} hidden layers $H_{i\in\{1;n_{\text{layers}}\}}$, each composed of n_{neurons_i} neurons $N_{i,j\in\{0;n_{\text{neurons}_i}-1\}}$ and an output layer O with n_{output} outputs. Each layer sends its outputs to the following layer until the output layer of the network is reached. Fig. 1 depicts a neural network with two hidden layers with four neurons each.

The layers are composed of a number of neurons computing a weighted sum of their inputs with the associated parameters called weights w_k and a bias B_j. The result is passed through an activation function f which sends data to the following layer according to the value of the output of the neuron o. The composition of a neuron is shown in Fig. 2. The most used activation function is ReLU (Rectified Linear Unit) where $a = f(o) = \max(o, 0)$.

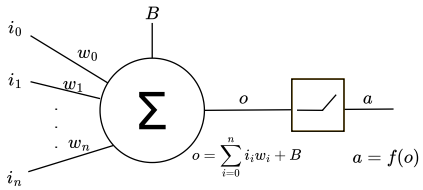


Fig. 2: Computation of a neuron followed by a ReLU activation function.

The number of layers and the number of neurons per layer are called the **architecture** of the network. The weights and bias of each neuron are called the **parameters**.

B. Dataflow accelerators

Dataflow accelerators are one of the possible ways to implement a neural network on an FPGA [5]. These implementations rely on matrix-vector multiplications to compute the neurons of the network. Each layer is handled by a dedicated computation unit as depicted in Fig. 3. These units store the weights of the neurons of the layer locally to avoid off-chip memory access and use Matrix-Vector-Threshold Unit to compute the neurons. Each computation unit can be customized at design time to optimize the overall accelerator performance. In particular, the level of parallelism of each layer can be configured by selecting the number of Processing Elements (PEs), which determines how many neurons are processed in parallel, as well as the number of SIMD lanes, which defines how many input channels are computed simultaneously. The inputs are sent from off-chip memory to the first computation unit when its buffer is empty.

The tailoring of the implementation to the neural network allows fast inferences and a higher throughput than sequential implementations. However the number PEs and SIMD lanes used for each layer directly impact the logic resources (registers and LUTs) on the chip which can be limited by its overall capacity. This allocation needs to be adapted regarding the implementation and application constraints.

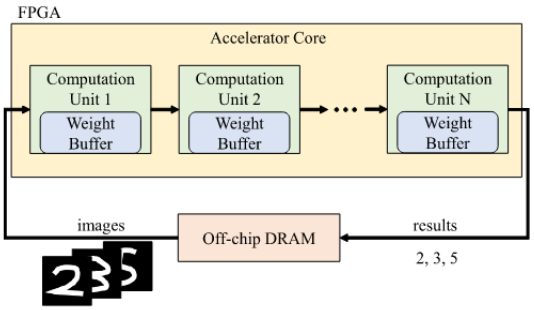


Fig. 3: Dataflow Accelerator. Extracted from [5].

Dataflow implementations require a huge design effort to tailor the implementation to the model. However frameworks like FINN [1] and hsl4ml [2] can be used to automatically generate a bitfile from a model which has been trained beforehand with a deep-learning framework like Pytorch.

C. Side-channel analysis on Neural networks

Side channel analysis to recover data from a neural network implementation is a growing topic in the literature. The most studied hardware target is CPU where input data [6], model architecture [4], [7]–[9] and weights of the neurons [6], [7], [10] can be recovered through power consumption or electromagnetic leakage. The attack methodologies typically rely on simple power analysis and correlation electromagnetic analysis.

On FPGA, the topic is much less studied. Some attacks aim to recover the architecture and parameters of the implemented neural network. Using side-channel analysis, the inputs [3], architecture [11]–[13] and weights [14] are recovered. These works focus on overlay implementations with a sequential computation of the layers. However the use of dataflow accelerators changes the computation of the network due to the possibility of parallelization and the pipelining of the layers. Existing attacks do not take into consideration the implementation specifities of these architectures. The implementation parameters (the number of PEs and SIMD lanes) represent novel information that could be recovered by an attacker who aims to reverse engineer the whole architecture. The recovery of these dataflow architectures parameters has been studied in few papers [15], [16]. Both of these attacks perform a profiled side-channel attack, using a profiling device to train a classifier to recover the dataflow implementation parameters and the architecture of the targeted network. This attacker model assumes that the attacker has access to an identical device and incurs a high data collection cost.

III. Attacker model

Attacker goal: The goal of the attacker is to reverse engineer the architecture of the network, *i.e.* the number of hidden layers and the number of neurons per layer, using power side-channel analysis.

Attacker knowledge: The attacker knows the size of the input data and the number of output classes. He knows the framework used to implement the accelerator. We consider a pipelined architecture of neural networks without parallelization inside a computation unit (PEs and SIMD lanes are set to 1). We consider that the number of neurons per layer is a power of two or multiple of ten with a minimum value of eight. This is a weak assumption given that commonly-used layers utilise these values. Furthermore, in the context of dataflow implementation, it is imperative that the number of neurons is a multiple of the number of PEs and SIMD lanes.

Attacker capability: During the attack, the attacker can trigger an inference of the neural network with an empty pipeline. He can measure the power consumption of the chip from the beginning to the end of the inference using a physical sensor. The sensor sampling frequency is a multiple of the FPGA clock frequency.

IV. Methodology

The attack presented in this work uses the power consumption traces measured to recover the number of hidden layers and the number of neurons per layer of the implemented neural network. The first step is to collect traces of consumption during the inferences. After obtaining the traces, the attack is done in two distinct parts:

- recovering the number of hidden layers and their timing using simple power analysis. This will be tackled in subsection B.
- recovering the number of neurons per layer using the measured timing of the layers. This will be explained in subsection C.

A. Collecting power consumption traces

The whole chip power consumption is measured during the inference. The sampling of the power consumption starts when the input data is sent to the accelerator and ends when the inference is done. This is a common practice in the context of side-channel attacks. The physical sensor should have at least the same clock frequency as the FPGA. N inferences are computed with different inputs and N power

consumption traces are collected. The number of samples of the traces depends on the duration of the inference and the sampling frequency. The N measured traces are then averaged to reduce the noise of the chip. By sending multiple inputs and averaging the traces of the power consumption, different patterns are observed corresponding to the computation of different layers. Using this approach, we can observe the pattern changes on the trace to determine the changes of layers.

B. Recovering the number of hidden layers

Given the pipelined computation of the layers of the network, the duration of the computation of a layer H_i is based only on n_{neurons_i}, the number of neurons in the layer and $n_{\text{neurons}_{i-1}}$, the number of neurons in its preceding layer H_{i-1} and takes $n_{\text{neurons}_i} \times n_{\text{neurons}_{i-1}}$. For example, a layer composed of 10 neurons with its preceding composed of 20 neurons, will need 200 cycles to be computed. It is independent of the value of its inputs or parameters.

The detection of the changes in the pattern is achieved through the computation of a matrix profile [17], a statistical tool used in time series data mining to detect anomalies, patterns and similarities. In the considered context, it is used to detect the pattern changes corresponding to the transitions from one layer to another. The matrix profile computes the Euclidean distance between a chosen subset, called window, and every other subset of the trace. Algorithm 1 presents the computation of the matrix profile. The size of the window has to be carefully chosen (see Section V) because a pattern smaller than the window size will not be detected.

Algorithm 1 Matrix profile computation.

N = number of samples
ws = window size
for i FROM 0 TO $N - ws$ **do**
  $window \leftarrow trace[i : i + ws]$
  for j FROM 0 TO $N - ws$ **do**
    $subset \leftarrow trace[j : j + ws]$
    $distance[j] \leftarrow \sqrt{\sum_{k=0}^{ws}(window[k] - subset[k])^2}$
  end for
  $matrixprofile[i] \leftarrow min(distance)$
end for

A low value of the matrix profile represents a recurrent pattern which corresponds to the computation of a layer, while a high value shows a window that never repeats, thus a transition between two distinct patterns. Hence, as a result of the matrix profile, we observe peaks in the values of the matrix profile corresponding to the transition of layers.

To locate the main peaks on the matrix profile, we use the prominence of the peaks which shows how significant a peak is compared to its surroundings. The computation of the prominence is presented in Algorithm 2 and depicted in Fig. 4.

Algorithm 2 Prominence computation.

for every points $\alpha(x_\alpha; y_\alpha)$ of $trace$ **do**
  Find next higher point on both sides
  Find lowest point (isolation) in between
  Keep the highest isolation $\delta(x_\delta; y_\delta)$
  P[x_α] = y_α - y_δ ▷ Vertical distance between α and δ
end for

A low value of prominence shows a point with the same height as its neighbors. A high value of prominence shows a point with a large vertical distance to the nearest isolation, corresponding to a local maximum. Using this detection of the peaks, we are able to segment the traces on the transitions of the layers. Given that each network is composed of an input layer and an output layer, the other transitions correspond to hidden layers. Thus we can recover the number of hidden layers in the network. The duration between two transitions also indicates the number of samples of each layer. Given that the clock frequency of the FPGA and the sampling frequency are known to the attacker, we can deduce the number of clock cycles of each layer. Using this methodology, we can obtain the number of hidden layers and the number of cycles $cycles\_meas_i$ for each layer H_i. We will use this information to find the number of neurons in each hidden layer.

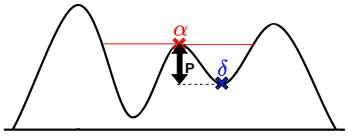


Fig. 4: Computation of prominence.

C. Recovering the number of neurons per layer

To recover the number of neurons per layer, we use the knowledge of the attacker model, *i.e.* the size of input and the number of output classes, and the information recovered in the first phase of the attack.

Given the pipelined computation of the layers, the number of cycles to compute a layer is determined only by the number of inputs of the layer and the number of neurons in the layer. For a given layer H_i the theoretical number of cycles is $cycles\_th_i = n_{inputs_i} \times n_{\text{neurons}_i}$. With the exception of the first hidden layer which directly receives the inputs of the network, the number of inputs of a layer H_i corresponds to the number of neurons in the previous layer $n_{\text{neurons}_{i-1}}$. The computation of $cycles\_th$ is depicted in Fig. 5.

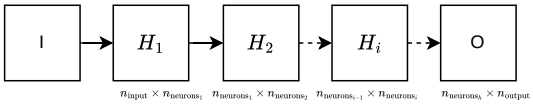


Fig. 5: Computation of the theoretical number of cycles per layer.

To recover the number of neurons in each layer, we look for the theoretical number of cycles that is the closest to the measured number of cycles collected in part B. Thus for each layer, we compute a hypothetical duration for a set value of n_{neurons} and look for the minimal error with the measured number of cycles. This can be expressed as an optimisation problem using the following constraints:

- n_{input}: size of the input layer of the network (Attacker model)
- n_{output}: size of the output layer of the network (Attacker model)
- n_{layers}: number of hidden layers (determined in Subsection B)
- $cycles\_meas = (meas_1, \ldots, meas_h)$ (determined in Subsection B)
- n_{neurons}: power of two or multiple of ten (Attacker model)

The goal is to recover the number of neurons in each layer, $n_{\text{neurons}} = (n_{\text{neurons}_1}, \ldots, n_{\text{neurons}_h})$, that minimizes the error between the hypothetical duration and the one deduced from side-channel measurements. This is expressed in Equation 1 where n_{neurons_0} is n_{input}.

TABLE I: Implemented architectures.

Victim #	Architecture of the network	n_{layers}	Inference duration (#Cycles)
1	784-20-20-20-10	3	17474
2	784-64-64-64-10	3	59802
3	784-8-32-32-32-16-10	5	10032
4	784-64-128-128-256-512-10	5	244506

$$\min \sum_{i=1}^{n_{\text{layers}}} \left| n_{\text{neurons}_{i-1}} \times n_{\text{neurons}_i} - cycles\_meas_i \right| \quad (1)$$

The vector n_{neurons} with the lowest error corresponds to the actual number of neurons of each layer of the network.

V. Experimental Setup and Results

A. Experimental Setup

The methodology described above was tested on several neural networks implemented on an FPGA. The targeted neural networks are MLPs composed of fully-connected layers, followed by a BatchNorm and a ReLU activation function. Table I presents the implemented architectures. All networks are quantized to 8 bits using Brevitas [18]. This is part of the FINN workflow to generate the accelerator.

The neural networks are trained on the MNIST [19] dataset made of handwritten digits from 0 to 9. The images in the dataset are 28 $\times$ 28 pixels in grayscale so the size of the input layer is 784 and the number of output classes is 10. The networks are trained on 100,000 training images during 20 epochs for a final test accuracy average of 97.5 %.

The dataflow accelerators are generated using FINN [1] a framework proposed by Xilinx (and AMD since 2022) to deploy quantized neural networks on FPGA. The number of PE and SIMD is set to 1 for each layer of the networks. For each layer, a theoretical number of cycles is given by FINN and reported in Table I.

The models are implemented on an Artix 7 FPGA target on a Chipwhisperer CW305 [20]. This board is designed for side-channel analysis and security evaluations. It is packed with a USB interface, facilitating data communication through a Python driver. The communication with the accelerator is made with a finite state machine that manages the inputs and outputs of the network. The transfer of an image to the accelerator is done when the previous image has been classified to keep the pipeline empty.

The power consumption traces are collected using the Chipwhisperer for the whole duration of the inference with a 10 MHz sampling frequency synchronized with the FPGA internal clock. For each configuration of neural network we compute 100 inferences with different inputs to collect 100 traces and average them together. Fig. 6 presents the example of 100 averaged traces for victim #3. The duration of the averaged traces is 9,181 samples, as shown in Fig. 6. The green dotted vertical lines show the theoretical changes of layers which are not known by the attacker. It is possible to observe the changes in the first layer, but the other layers are not distinguishable by sight.

The experimental results are presented in Subsection B with a focus on victim #3 (as expressed in Table I) as a case study. The results obtained from the other networks are discussed in Subsection C.

B. Case study on victim #3

The matrix profile is computed using the Python library Stumpy [21]. The size of the window has a strong impact on the peaks created by the matrix profile. This impact is depicted in Fig. 7

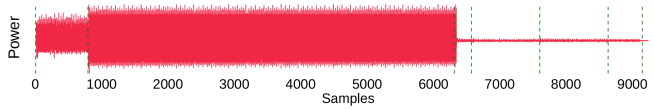
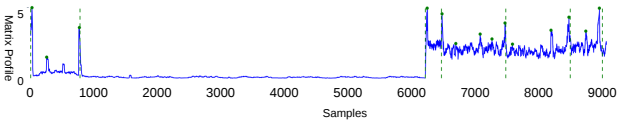
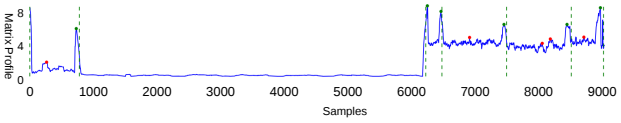


Fig. 6: 100 averaged traces sampled with the Chipwhisperer during the inferences of the victim #3. The green dotted vertical lines show the theoretical changes of layers given by FINN (unknown to the attacker).

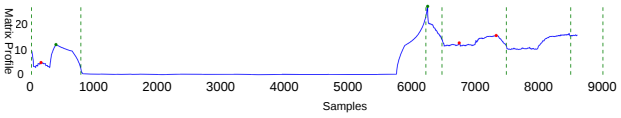
where it is computed with window sizes of 30, 64 and 600 samples. A window size too large will create flattened peaks and will fail to detect the layers of a smaller size. This is depicted in Fig. 7c where the third hidden layer composed of 20 neurons is computed in 400 samples and cannot be detected by a matrix profile window of 600. Moreover the peaks created would flatten out and their position would not be accurate. This offset is clearly visible by comparing Fig. 7b and 7c. A window size that is too small will reduce the value of the distance, and the peaks created by the changes in pattern will not have a significant enough prominence to stand out from the noise. This is shown in Fig. 7a. Thus the matrix profile window size depends on the minimal layer length however this information is not available to the attacker. The total length of the traces can be used to formulate a hypothesis. In accordance with the attacker model, the minimum number of neurons in a given layer is 8. Hence, the minimal number of cycles to compute a layer is 64. This value is used as window size in the attack as depicted in Fig. 7b. It is important to note that the matrix profile is computed on $N - window\_size$ samples. If the window size is too large, the output layer will not be covered by the range of the matrix profile and it will not be detected. This is depicted in Fig. 7c.



(a) window size of 30.



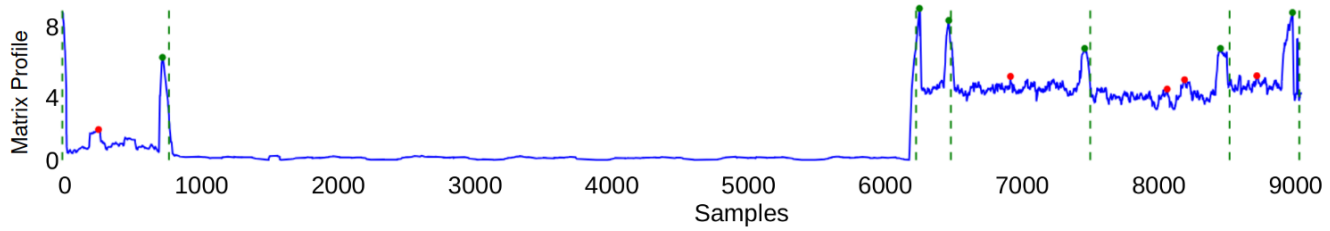
(b) window size of 64.



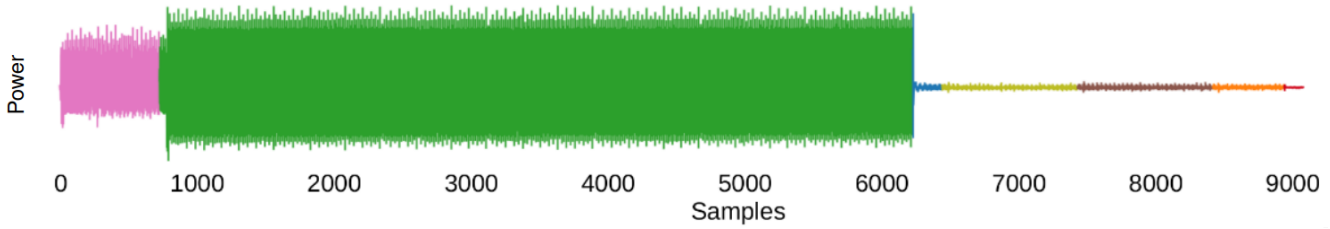
(c) window size of 600.

Fig. 7: Matrix profile on the averaged traces with window sizes of 30 (a), 64 (b), 600 (c) samples. Green dotted vertical lines illustrates the theoretical change of layer.

The peaks are detected using the `find_peaks` function of the open source Python library SciPy [22], whose parameters are the minimum distance between two consecutive peaks and the peak prominence. The minimum distance between two consecutive peaks is set to 64. The prominence is selected on the basis of its distribution. It is estimated that the distribution is approximately Gaussian of



(a) Peaks selected using the prominence.



(b) Segmented traces of victim #3 using the matrix profile.

Fig. 8: Cutting of the traces using the prominence of each peak.

TABLE II: Recovered architecture of the victim #3. First column in green is not known by the attacker and only included for comparison purposes. Gray cells are fixed values, white cells are values determined by the resolution of the optimisation problem. Bold values are known by the attacker.

n	$cycles\_meas$	$n_{\text{neurons}_i} \times n_{\text{neurons}_{i-1}}$	$cycles\_error$	Recovered n
784	707	784	71	**784**
8	6301	6272	29	8
32	201	256	55	32
32	1008	1024	16	32
32	1021	1024	3	32
16	506	512	6	16
10	159	160	15	**10**

mean μ and standard deviation σ. We detect outliers, typically above the 2σ bound, corresponding to the changes of layers. The matrix profile values are shown in Fig. 8a. Green and red dots indicate high-prominence peaks which are selected and undesired peaks which have a low prominence and are not selected respectively. The layers are then segmented using the position of the peaks. Fig. 8b shows the segmentation results on victim #3 with seven sub-traces corresponding to the input layer, five hidden layers and the output layer. This corresponds to the actual neural network.

To recover the number of neurons per layer, we solve the optimisation problem exposed in Section IV.C. We use the Python open source library CPMpy [23] to solve this optimisation problem. The solver is based on Google OR-Tool constraint programming solver. Here are the constraints applied for the victim #3 case study:

- n_{input} = 784 (Attacker model)
- n_{output} = 10 (Attacker model)
- h = 5 (determined in Subsection B)
- $cycles\_meas = (707; 6301; 201; 1008; 1021; 506; 159)$ (determined in Subsection B)

With these constraints, the lowest sum of errors computed in Equation 1, is 124 and corresponds to the correct architecture. Thus we successfully recover the number of neurons in all five layers. Table II presents the recovered architecture and the error in the number of cycles for each layer. The ten first proposed architecture candidates are shown in Table III.

TABLE III: Top 10 architectures candidates of the solver ordered by sum of errors.

Rank	Architecture candidates	$\min(\sum cycles\_error)$
1	**8-32-32-32-16**	**124**
2	8-30-32-32-16	140
3	8-32-32-30-16	202
4	8-32-30-32-16	214
5	8-30-32-30-16	218
6	8-30-30-32-16	258
7	8-32-32-30-20	260
8	8-30-32-30-20	292
9	8-32-32-32-20	294
10	8-32-30-30-16	300

C. Experimental results on other victims

This methodology recovers the exact number of layers in all tested architectures. Table IV shows all the recovered architectures. The segmentation may differ by a few samples from the theoretical number given by FINN. Therefore, the number of cycles recovered has a negligible margin of error, with a maximum discrepancy of 10%. This error is of negligible significance and the does not affect the recovery of the number of neurons. Smaller layers (8-20 neurons) show more subtle differences; however, the transitions are still detectable using the matrix profile. This is demonstrated for the last layers of the victim #1 and illustrated in Fig. 9. Changes between two large layers are more easily detectable, resulting in clear and prominent peaks in the matrix profile that can be easily identified. This is illustrated for the victim #4 in Fig. 10. and Fig. 11. The minimum sum of errors always corresponds to the correct architecture. It is clearly smaller than other architecture candidates as shown in the first column of Section C in Table IV.

VI. Conclusion

This paper presents the first unsupervised architecture-recovery attack on neural networks with a dataflow implementation on FPGA. Using only power consumption traces, we demonstrate that it is possible to recover the number of layers, their timing and the number of neurons in each layer. This approach needs no training on a profiling device. This attack was successfully tested on several

TABLE IV: Recovered architectures

	Part B : recovering the number of layers and their duration			Part C : recovering the number of neurons		
Victim #	Recovered n_{layers}	Mean error on duration (#Samples)	σ error on duration (#Samples)	min Σ error for Top 3 architecture candidates	Recovered architecture	Accuracy
1	**3**	30.7	8.7	**156**; 196; 230	**784-20-20-20-10**	100%
2	**3**	19	13.9	**191**; 365; 591	**784-64-64-64-10**	100%
3	**5**	20.8	4.1	**124**; 140; 202	**784-8-32-32-32-16-10**	100%
4	**5**	106.6	136.2	**1063**; 4159; 9745	**784-64-128-128-256-512-10**	100%

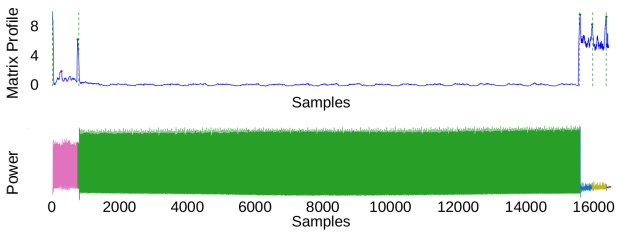


Fig. 9: Segmentation of the traces of the victim #1.

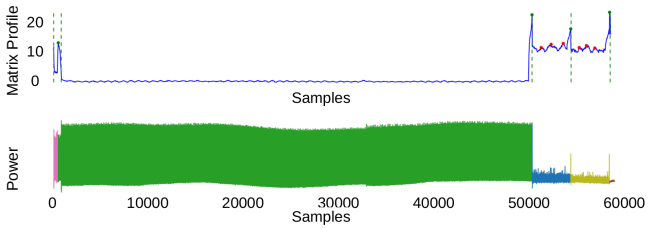


Fig. 10: Segmentation of the traces of the victim #2.

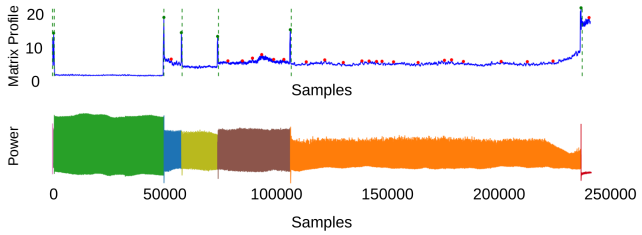


Fig. 11: Segmentation of the traces of the victim #4.

architectures with different numbers and size of layers implemented using FINN. Since the proposed attack is based on the computation time of the layers, it is susceptible to countermeasures such as random time operations which need to be investigated. Future work could explore other architectures with a different number of PEs and SIMD lanes for each layer.

References

[1] Y. Umuroglu, N. J. Fraser *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA*, 2017, pp. 65–74.

[2] F. Fahim, B. Hawks *et al.*, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," *CoRR*, vol. abs/2103.05579, 2021.

[3] L. Wei, B. Luo *et al.*, "I know what you see: Power side-channel attack on convolutional neural network accelerators," in *Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA*, 2018, pp. 393–406.

[4] R. Joud, P. Moëllic *et al.*, "Like an open book? read neural network architecture with simple power analysis on 32-bit microcontrollers," in *Smart Card Research and Advanced Applications 2023, Amsterdam, The Netherlands, 2023, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 14530, 2023, pp. 256–276.

[5] F. Hamanaka, T. Odan *et al.*, "An exploration of state-of-the-art automation frameworks for fpga-based DNN acceleration," *IEEE Access*, 2023.

[6] S. Maji, U. Banerjee *et al.*, "Leaky nets: Recovering embedded neural network models and inputs through simple power and timing side-channels - attacks and defenses," *IEEE Internet Things J.*, vol. 8, no. 15, pp. 12 079–12 092, 2021.

[7] L. Batina, S. Bhasin *et al.*, "CSI NN: reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA*, 2019, pp. 515–532.

[8] E. Malan, V. Peluso *et al.*, "Enabling DVFS side-channel attacks for neural network fingerprinting in edge inference services," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2023.

[9] Y. Won, S. Chatterjee *et al.*, "Deepfreeze: Cold boot attacks and high fidelity model recovery on commercial edgeml device," in *IEEE/ACM International Conference On Computer Aided Design*, 2021.

[10] C. Gongye, Y. Fei *et al.*, "Reverse-engineering deep neural networks using floating-point timing side-channels," in *57th ACM/IEEE Design Automation Conference*, 2020.

[11] Y. Zhang, R. Yasaei *et al.*, "Stealing neural network structure through remote FPGA side-channel analysis," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 4377–4388, 2021.

[12] H. Yu, H. Ma *et al.*, "Deepem: Deep neural networks model recovery through EM side-channel information leakage," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust, San Jose, CA, USA, 2020*. IEEE, 2020, pp. 209–218.

[13] S. Tian, S. Moini *et al.*, "A practical remote power attack on machine learning accelerators in cloud fpgas," in *Design, Automation & Test in Europe Conference & Exhibition*, 2023.

[14] K. Yoshida, M. Shiozaki *et al.*, "Model reverse-engineering attack against systolic-array-based DNN accelerator using correlation power analysis," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 2021.

[15] V. Meyers, D. Gnad *et al.*, "Reverse engineering neural network folding with remote FPGA power analysis," in *30th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, New York City, NY, USA, 2022*. IEEE, 2022, pp. 1–10.

[16] G. Lomet, R. Salvador *et al.*, "Side-channel extraction of dataflow AI accelerator hardware parameters," in *31st IEEE International Symposium on On-Line Testing and Robust System Design, Ischia, Italy, 2025*. IEEE, 2025, pp. 1–7.

[17] C. M. Yeh, Y. Zhu *et al.*, "Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *IEEE 16th International Conference on Data Mining, ICDM*, 2016.

[18] G. Franco, A. Pappalardo *et al.*, "Xilinx/brevitas," 2025.

[19] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, 2012.

[20] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *Constructive Side-Channel Analysis and Secure Design , 2014, Paris, France*, ser. Lecture Notes in Computer Science, vol. 8622, 2014, pp. 243–260.

[21] S. M. Law, "STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining," *The Journal of Open Source Software*, vol. 4, no. 39, p. 1504, 2019.

[22] P. Virtanen, R. Gommers *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, 2020.

[23] T. Guns, "Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example," in *Modref*, 2019.