# Deep-learning Model Extraction through Software-based Power Side-channel

Xiang Zhang[1], Aidong Adam Ding[2], Yunsi Fei[1]
[1]*Department of Electrical and Computer Engineering,* [2]*Department of Mathematics*
*Northeastern University*, Boston, MA, USA
{zhang.xiang1, a.ding, y.fei}@northesatern.edu

***Abstract*—Deep learning (DL) techniques have been increasingly applied across various applications, facing a growing number of security threats. One such threat is model extraction, an attack that steals the Intellectual Property of DL models, either by recovering the same functionality or retrieving high-fidelity models. Current model extraction methods can be categorized as learning-based or cryptanalytic, with the latter relying on model queries and computational methods to recover parameters. However, these are limited to shallow neural networks and are computationally prohibitive for deeper DL models.**

**In this paper, we propose leveraging software-based power analysis, specifically the Intel Running Average Power Limit (RAPL) technique, for DL model extraction. RAPL allows us to measure power leakage of the most popular activation function, ReLU, through a software interface. Consequently, the ReLU branch direction can be leaked in the software power side-channel, a vulnerability common in many state-of-the-art DL frameworks. We introduce a novel methodology for model extraction Algorithm from input gradient assisted by side channel information. We implement our attack on the oneDNN framework, the most popular library on Intel processors. Compared to prior work, our model extraction, assisted by the software power side-channel, only requires 0.8% of the queries to retrieve a 5-layer MLP. We also successfully apply our method to a common Convolutional Neural Network (CNN) - Lenet-5. To the best of our knowledge, this is the first work that extracts CNN models with more than 5 layers based solely on queries and software.**

***Index Terms*—Model Extraction, RAPL Power Side-Channel, Input Gradient**

## I. Introduction

Deep learning has been widely applied in various application domains. However, deep neural network (DNN) models and implementations are facing increasing threats. The DNN model, including the structure, hyperparameters, and parameters, is becoming valuable intellectual property (IP) and adversaries aim to steal the IP [1]–[3].

In this paper we focus on model extraction, a common IP-stealing attack. Generally, there are three types of model extraction: learning-based model extraction and cryptanalytic model extraction. Learning-based model extraction is training an approximate model by querying inputs and corresponding outputs. Cryptanalytic model extraction is analyzing DNN model inputs and outputs by traditional statistical and cryptanalytic methods. Comparing to learning-based model extraction, cryptanalytic model extraction is more apparent because of the black-box attribute of machine learning approaches. Previous cryptanalytic model extraction work [4], [5] recover the oracle model with high fidelity, i.e., achieving functional equivalence. They are software methods, and rely on well-guided queries to find witnesses, which are model inputs that make the input to the ReLU activation function of a neuron zero, and then *calculate* the model parameters based on the witnesses.

The shortcoming of the previous software methods [4], [5] is their scalability to deeper and more complex models. This is because they only use controlled inputs to query the model and check the logits (the model outputs) so as to find a witness. However, their methods are unable to associate the witness with a specific neuron directly and therefore result in exponential computational complexity.

In this work, we propose the use of a software-based power analysis tool, the Running Average Power Limit (RAPL) technique, to facilitate the extraction of deep learning models. This tool enables us to directly measure the power leakage of the ReLU function via the RAPL software interface, thereby obtaining the branch direction. Coupled with an innovative model extraction algorithm that relies on input gradients from activation patterns, we can accelerate the model extraction process. By collecting output from queries and RAPL results, we can recover model parameters and apply them in a rapid and efficient model extraction algorithm. In comparison with previous work, our approach to model extraction, assisted by software power side-channels, requires significantly fewer queries.

This paper contributes to the field in the following ways:

- Firstly, we conduct a comprehensive analysis of all ReLU implementations in modern Machine Learning (ML) frameworks, including Pytorch, Tensorflow, oneDNN, and Darknet. We find that all these implementations are susceptible to software-based power side-channel attacks.
- Secondly, we present a practical and accurate method to extract the model activation pattern from the software-based side channel.
- Lastly, we develop a model extraction algorithm, which hinges on the output from queries and their corresponding activation patterns. Our approach leverages the activation pattern data gathered from the software-based power side-channel, considerably reducing the number of required queries and facilitating the process of model extraction.

The rest of this paper is arranged as follows. In Section II, we provide an introduction to power side-channels and the process

This work was supported in part by National Science Foundation under grants CNS- 2212010 and SaTC-1929300.

of DNN model extraction. Next, in Section III, we discuss the role of the software power side-channel in the context of DNN model activation functions. Our novel model extraction algorithm is then presented in Section IV. Finally, we evaluate our method and discuss the results in Section V.

## II. Background

In this section, we first introduce background of DNN model extraction and software-based power side-channel. We then present various machine learning frameworks and their ReLU implementations.

### A. Deep Neural Network Model Extraction

Deep neural network model extraction is a type of attack which aims at recovering the oracle model. Most of prior works are focusing on *accuracy*, i.e., the extracted has similar performance as the oracle over the entire dataset. However, Jagielski etc. [4] argue that *fidelity* is equally important to *accuracy*, where *fidelity* measures the agreement on any input data. They have extracted 2-layer Multi-layer Perceptrons (MLP) with high accuracy and high fidelity. In general, model extraction is classified into two types:

1) **Learning-based model extraction**: Learning-based model extraction works by submitting queries to the oracle model and using machine learning on the query results to generate an approximate copy. This method is particularly beneficial when dealing with large neural networks, where a more detailed extraction process might be impractical due to the model's size. In general, learning-based model extraction can achieve high accuracy, as demonstrated by several studies [6], [7]. Some works even aim for high fidelity [4], i.e., extracted results not only work similarly to the original model but also closely mirror its internal structure and parameters. However, it's worth noting that a learning-based extraction may result in a model that, while functionally similar to the oracle model, has a different structure and parameters. This distinction could impact the model's behavior in edge cases or when presented with data that differ from the original training set.
2) **Cryptanalytic model extraction**: Cryptanalytic model extraction seeks to recover an exact copy of the oracle model, aiming to reproduce its outputs with high fidelity across all possible inputs from the dataset. This is generally more challenging than learning-based model extraction because it involves recovering parameters one by one, similar to the byte-by-byte key recovery process in cryptanalytic attacks. Carlini et al. [5] were among the first to formalize DNN model extraction as a cryptanalytic problem. They devised special inputs that put the model execution in a certain state and used equation solving to recover the model parameters. Their work focused on the ReLU activation function, which is the most commonly used in modern DNNs. Other researches [8], [9] also aim to extract model parameters with the help of side-channel information. These methodologies illustrate the varying strategies that can be utilized to achieve high fidelity model extraction.

Cryptanalytic model extraction has inherent drawbacks: specifically, its utility diminishes as the complexity and depth of the neural network increase, with the number of necessary queries rising exponentially. This is primarily due to the challenges in linking a specific input (referred to as a witness) to the corresponding neuron or layer in a critical state.

In our research, we've addressed this limitation through a two-pronged approach, namely, leveraging a software power side-channel and implementing an input gradient-based model extraction method. The side-channel provides valuable insights into the activation pattern of each layer in the neural network. In addition, the new model extraction method we've developed has consistent complexity across different layers, helping us reduce the number of queries significantly for model extraction.

### B. Power Side-channel and Running Average Power Limit (RAPL)

Power side-channel analysis has been a popular and effective attack against cryptographic implementations [10]. It leverages the data dependence of power consumption of CMOS circuits to retrieve the secret. Over the past decade, many attacks and countermeasures have been proposed on various cipher systems. However, collecting power traces from a victim system relies on hardware equipment and proximity/contact with the device. Modern Intel processors have incorporated Running Average Power Limit (RAPL) technique which includes on-chip power sensors and corresponding Model-specific Registers (MSRs) [11]. Software interfaces are provided for users to read the accumulated energy consumption of various power domains. Recently, Lipp et al. [12] have utilized RAPL to perform software-based power side-channel analysis. It is demonstrated that RAPL has sufficient resolution for practical power attacks on common ciphers, including the RSA algorithm in Mbed TLS and AES-NI implementations. To measure the power consumption of target instructions, they employ the zero-stepping and single-stepping [13] in Intel SGX enclave to repeat instruction executions. Our work utilizes RAPL power analysis of DNN model execution to extract the model.

The Intel RAPL technique provides four power domains for both measuring the energy consumption and throttling them at run-time [11].

- *Package (PKG)*: measures the energy consumption of the entire processor, including all cores, the integrated GPUs and last level cache and memory controller.
- *Power Plane 0 ($PP_0$)*: measures energy consumption of all cores.
- *Power Plane 1 ($PP_1$)*: measures energy consumption of on-chip GPUs.
- *DRAM*: this domain measures energy consumption of the DRAM.

Intel SGX (Software Guard Extensions) is an instruction set extension to provide Trusted Execution Environment (TEE) for both code and data [14]. The SGX-Step [13] is an attack framework to control instruction execution in SGX through interrupts. In particular, **Zero-stepping** can repeat an instruction while **Single-stepping** moves on to the next instruction. We

use these mechanisms to run instructions repetitively in order to magnify the power consumption of a specific piece of code, similar to the prior work [12].

### C. ReLU Activation and its Implementation

Contemporary deep learning frameworks are furnished with all vital elements that aid in building specialized deep neural networks. A critical component of these networks is the activation function, which imparts non-linear properties to the model, enabling it to learn from complex data. Prominent among these activation functions is the **Rectified Linear Unit** (ReLU) [15]. ReLU is the most frequently utilized activation function in deep learning, renowned for its simplicity and efficiency. The ReLU function is defined as a piece-wise linear function, as follows:

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

For easy usability, frameworks such as Pytorch [16] and Tensorflow [17] are the frond-end to accept Python programs for model configurations. For high efficiency, frameworks such as Darknet [18] for embedded systems and Intel oneDNN [19] for general-purpose systems are C/C++ based and provide the back-end. Pytorch and Tensorflow are integrated with oneDNN for outputting efficient DNN models on Intel processors. Our RAPL-based power side-channel attack targets the common oneDNN library while Darknet is also vulnerable to power side-channel attacks.

There are several variants of ReLU activations, including Exponential Linear Unit (ELU) [20] and Leaky ReLU and Parametric ReLU (PReLU) [21]. They are all implemented with similar functions in modern DNN frameworks, as shown below:

$$f(x) = \begin{cases} x * \alpha & x < 0 \\ x & x \geq 0 \end{cases}$$

$\alpha$ is a special multiplier for the negative branch. For the targeted oneDNN library, Listing 3 shows the ReLU implementation in C with a conditional branch that has two clauses. *alpha* is usually set to zero. Listing 4 shows the assembly code, where two jump instructions (`ja` and `jmp`) form two branches clauses in ReLU. Such imbalanced branch clauses consume different power and form a power side-channel. For Darknet, Listing 1 gives the ReLU implementation. Although it does not contain a conditional branch, as shown in Listing 2, the power consumption is dependent on whether $x > 0$ or not, leaking the direction of the ReLU function in the power side-channel.

Listing 1: Darknet ReLU

```
relu(x)

{return x*(x>0);}

// End
```

Listing 2: Darkent ReLU in assembly

```
pxor      %xmm1,%xmm1
movaps    %xmm1,%xmm2
movss
0xf63(%rip),%xmm1
cmpltss   %xmm0,%xmm2
andps     %xmm2,%xmm1
mulss     %xmm1,%xmm0
cvtss2sd  %xmm0,%xmm0
```

Listing 3: oneDNN ReLU

```
relu(x, alpha
    )

{return

(x > 0)?

 x:
 x*alpha;}
// End
```

Listing 4: oneDNN ReLU in assembly

```
71e3ed: movaps  %xmm3,%xmm4
71e3f0: movaps  %xmm4,%xmm0
71e3f3: add     $0x18,%rsp
71e3f7: retq
...
71e455:
ucomiss 0xN(%rip),%xmm0
71e45c: ja      71e3ed
71e45e: movaps  %xmm3,%xmm4
71e461: mulss   %xmm1,%xmm4
71e465: jmp     71e3f0
```

### D. ReLU Activation Pattern and Input Gradient

In a ReLU-based model, the direction of ReLU plays a critical role in the forward pass. The **activation pattern** of a ReLU neuron network is represented as vectors that reveal the direction of the ReLU condition: $\mathbf{A}_i = [a_{i1}, a_{i2}, \ldots, a_{ih_i}]$ for the $i$-th layer with $h_i$ neurons. Here, for the $j$-th ReLU neuron on the $i$-th layer, $a_{ij} = 1$ indicates activation (i.e. it has positive output because its input value is greater than 0) while $a_{ij} = 0$ indicate its inactivation.

The **Input gradient** is a concept that illustrates how minute variations in the input can impact the output of a model. It serves as a crucial tool for understanding model behavior [22], implementing adversarial attacks [23], [24], and visualizing features [25], [26]. In this work, we use $\mathbf{I}_k$ to denote the **input gradient** vector consisting of derivatives of the network output $\mathbf{Y}$ with respective to the $k$-th component of the input vector $\mathbf{X} = [x_1, \ldots, x_n]$. In other words, the input gradient $\mathbf{I}_k$ is the $k$-th row of the Jacobian matrix for the network.

Using side-channel measurements, we can obtain the activation pattern. We propose a fast and efficient way of recover the DNN ways through comparison of the input gradients for special pairs of inputs following certain activation patterns.

## III. ReLU Branch Detection via RAPL Leakage

In this section, we analyze the ReLU branch execution through RAPL side-channel leakage. The root cause of leakage is the two conditions of a ReLU execution vary in energy consumption, which can be read through RAPL registers. We use *energy consumption* and *power consumption* interchangeably if the measuring duration is fixed. The software power side-channel not only leaks the ReLU direction, but also allows us to locate the neuron with the specific ReLU.

### A. Attacker Model

We consider an attack scenario where the victim model is running in Intel SGX enclaves for confidentiality protection [14], as the OS can be malicious too. The attacker only knows the structure of the DNN model as many are commonly used, but has no knowledge of the model parameters with the enclave protection. The attacker also knows which deep learning framework is in use, such as Pytorch or Tensorflow. The adversary can only query the model and can access Intel RAPL's MSRs.

### B. Experimental Setup and the RAPL Power Side-channel on ReLU

Our experimental platform is a workstation with Intel i7-7700 processor (KabyLake architecture) and Ubuntu 18.04.1. The Linux kernel version is 5.4.0. In this work, we observe $PP_0$ (power consumptions of the cores), because computations including the ReLU branch execution are only on cores. In our test machine, the $PP_0$ energy consumption updating rate is around 50 $\mu s$, similar to the prior work [12].

However, such an update rate is insufficient to capture the energy consumption difference of the ReLU function execution under two different conditions. We design experiments to repeat ReLU executions long enough to magnify the power side-channel. We read the RAPL MSRs before and after the repeated ReLU functions to measure the energy consumption.

As we discussed in Section II-C, the ReLU implementations are different in different frameworks. For the oneDNN implementation shown in Listing 3, the negative value of $x$ will result in more energy consumption due to the floating point multiplication ($alpha * x$). Fig. 1 shows the kernel distribution estimation of ReLU branches (repeat ReLU instructions for 10B times) energy consumptions, where positive and negative stand for $x \geq 0$ and $x < 0$, respectively.

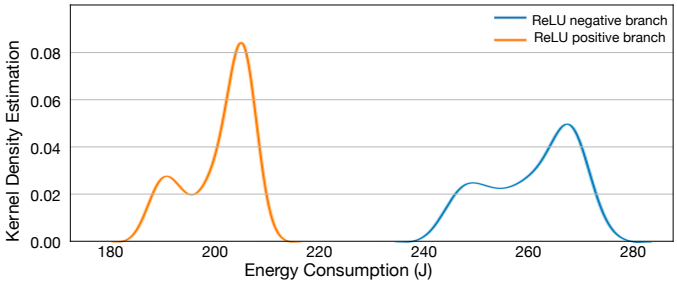


Fig. 1: Kernel Distribution Estimation of Energy Consumption (in J) of ReLUs in oneDNN

Similarly for the Darknet implementation, Fig. 2 shows that we can identify the positive or negative branch distinctly, and interestingly the negative branch still consumes more energy even though it is multiplied with zero.

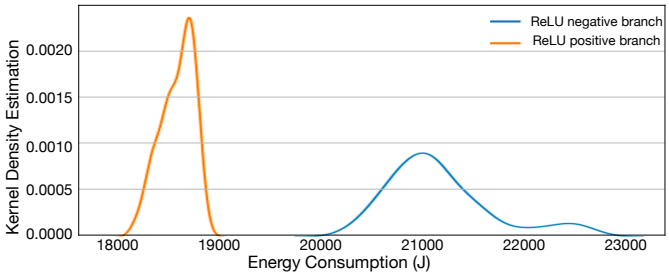


Fig. 2: Kernel Distribution Estimation of Energy Consumption (in J) of ReLUs in Darknet

### C. Automatic Repeating with SGX-Step

The previous results are obtained for plain repetitions of instructions (put in a loop). To automate repeated executions of selected ReLU functions, we adopt the SGX-Step technique similar to the prior work [12]. Fig. 3 shows the Energy Kernel Distribution Estimation of oneDNN ReLU implementation in Intel SGX enclaves, while only repeated for 10M times. Compared to the results shown in Fig. 1, the power consumption is much higher, even though for 1000 times less repetitions. This is due to the power overhead of SGX enclaves, including heavy cryptographic operations and memory accesses. Note the two distributions have more overlap, and the positions of them are opposite to those in Fig. 1: here the positive branch execution consumes more energy. We attribute this to complex operations in SGX.

Although the power side-channel is weaker in SGX (because of the overlap), it still exists for ReLU executions. We set $1348J$ as the threshold to distinguish the two distributions. With such a threshold, we have 10% inaccuracy for the negative branch and 7% inaccuracy for the positive branch. We keep measuring one neuron three times and average the energy consumption. The error rate drops to 0.1%, and we can guarantee the correctness of branch's direction.

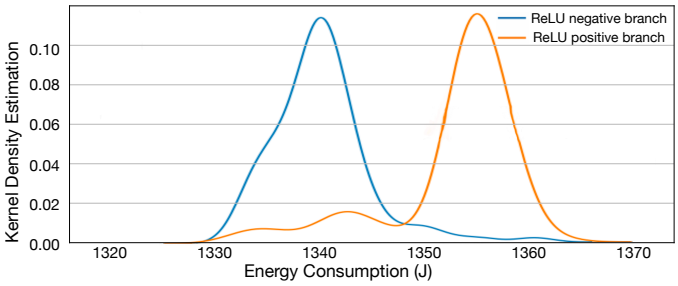


Fig. 3: Kernel Distribution Estimation of Energy Consumption of ReLU repeating 10M in oneDNN in Intel SGX

## IV. Input Gradient-based Model Extraction

In this section, we present a efficient method for extracting model parameters of deep neural networks, including multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs), based on input gradient and the RAPL-based power side channel.

Previous methods focused on identifying witnesses, which is the input values to the network that correspond to ReLU critical points. These witnesses were then used to recover weights through algebraic equations. Using the RAPL side-channel, we can find the witness faster with much fewer queries. Also since we can use RAPL side-channel to obtain the network's activation patterns, we recover the weights through simpler algebraic equations using input gradients. Overall, our approach offers a more efficient and effective means of extracting model parameters from deep neural networks.

### A. Terminology

Previous works have introduced methods for querying the model and inferring the parameters from corresponding inputs and outputs [4], [5]. Attackers can use any types of inputs, whether real or arbitrary, and feed them to the oracle model to obtain the outputs. This type of model extraction is known as query-based model extraction.

To find the witness for a target ReLU neuron, the previous works search for a pair of queries with inputs very close to each other and differs in the activation of this neuron. Then the

witness is known for precision up to the difference between this pair of inputs. We will utilize similar pairs of queries, called **Flip Pairs**, whose two inputs lead to different activation for the target ReLU neuron but have the same activation pattern for all other neurons in the network. The flip pairs usually can be found for precision much lower than required for the witness search, thus can be found faster.

We propose a fast way to recover the weights through comparison of the input gradients for two inputs in a flip pair. The input gradient can be calculated by comparing the outputs for two queries with the same activation pattern. We call a pair of queries with the same activation pattern as *Activation-Inclusive Pair*, and a pair of queries with different activation patterns as *Activation-Exclusive Pair*. For example, all flip pairs are activation-exclusive pairs.

### B. General Model Extraction Framework

A property of the input gradient is that it only depends on the activation pattern $\mathbf{A}$ and the weights $\mathbf{W}$ but not the biases $\mathbf{B}$. Thus we recover the weights and biases in separate steps. Firstly, we find input gradients for using methods described in the later section IV-C2. Secondly, by comparing input gradients of flip pairs, we easily find the last layer weights up to a positive multiplicative constant. As shown in prior works [4], [5], recovering weights in each layer up to an unknown positive multiplicative constant is sufficient to recover a network that is functionally equivalent to the original network. Thirdly, knowing weights in later layers, we iterate the process layer by layer to recover weights for earlier layers. Lastly, with the recovered weights, we can build equations from the input and output of flip pairs to recover the biases.

We illustrate the model extraction process on a toy example described next.

*1) A Toy Example and Notations:* We illustrate our algorithm using a toy example. Here we introduce the example and notations. Figure 4 shows the 3-4-5-2 MLP of four layers with heights $(h_0, h_1, h_2, h_3) = (3, 4, 5, 2)$. That is, the MLP has a 3-dimensional input $\mathbf{X} = [x_1, x_2, x_3]$, two hidden layers with 4 and 5 ReLU neurons respectively, and the 2-dimensional output $\mathbf{Y} = [y_1, y_2]$. The output $\mathbf{Y}(\mathbf{X})$

$$= [[[(\mathbf{X}\mathbf{W_1}+\mathbf{B_1})*\mathbf{A_1}(\mathbf{X})]\mathbf{W_2}+\mathbf{B_2}]*\mathbf{A_2}(\mathbf{X})]\mathbf{W_3}+\mathbf{B_3} \quad (1)$$

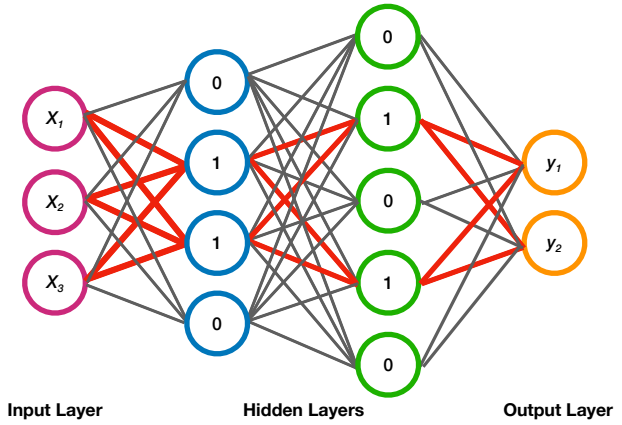


Fig. 4: An Example of Weights Recovery in MLP

The weight matrices for the three layers are $\mathbf{W_1}$, $\mathbf{W_2}$ and $\mathbf{W_3}$ of sizes respectively $3 \times 4$, $4 \times 5$ and $5 \times 2$. We denote $w_{ijk}$ as the weight connecting to the $k$-th neuron in the $i$-th layer from the $j$-th neuron in the prior $(i-1)$-th layer. For example, the first layer weight matrix $\mathbf{W_1}$ is

$$\begin{bmatrix} w_{111} & w_{112} & w_{113} & w_{114} \\ w_{121} & w_{122} & w_{123} & w_{124} \\ w_{131} & w_{132} & w_{133} & w_{134} \end{bmatrix}$$

The Jacobian matrix consists of all input gradient vectors as row vectors: $\mathbf{J} = [\mathbf{I}_1^T, \mathbf{I}_2^T, \mathbf{I}_3^T]^T$ here. For a MLP using identity activation function, which is a simple linear system, the Jacobian is simply the product of the weights matrices of each layer: $\mathbf{J} = \mathbf{W}_1 \cdot \mathbf{W}_2 \cdots \mathbf{W}_D$ for a network with depth $D$. For a MLP with ReLU neurons, an inactivate ReLU zeros out the connected weights in the gradient, thus

$$\mathbf{J} = (\mathbf{W}_1 * \mathbf{A}_1) \cdots (\mathbf{W}_{D-1} * \mathbf{A}_{D-1}) \cdot \mathbf{W}_D \quad (2)$$

where $*$ denotes the broadcasting operation so that $\mathbf{W}_i * \mathbf{A}_i = [\mathbf{W}_{i,1}, ..., \mathbf{W}_{i,h_i}] * [a_{i,1}, ..., a_{i,h_i}] = [\mathbf{W}_{i,1} a_{i,1}, ..., \mathbf{W}_{i,h_i} a_{i,h_i}]$. Here $\mathbf{W}_{i,j}$ denotes the $j$-th column of the weight matrix $\mathbf{W}_i$.

Notice that all neurons in last output layer are not ReLU type, and are always activated. Thus the last layer weights are not modified by any activation vector in the Jacobian formula (2). The input gradient $\mathbf{I}_k$ is the $k$-th row of the Jacobian matrix:

$$\mathbf{I}_k = (\mathbf{W}_{1,k} * \mathbf{A}_1) \cdot (\mathbf{W}_2 * \mathbf{A}_2) \cdots (\mathbf{W}_{D-1} * \mathbf{A}_{D-1}) \cdot \mathbf{W}_D. \quad (3)$$

For the toy example, the third input gradient would be $\mathbf{I}_3 = (\mathbf{W}_{1,3} * \mathbf{A}_1)(\mathbf{W}_2 * \mathbf{A}_2)\mathbf{W}_3 = ([w_{131}, w_{132}, w_{133}, w_{134}] * \mathbf{A}_1)(\mathbf{W}_2 * \mathbf{A}_2)\mathbf{W}_3$.

For a specific input $\mathbf{X}$, Figure 4 shows the activation pattern (the red colored weights are on the activated paths) with values

$$\begin{aligned} \mathbf{A} = [\mathbf{A}_1, \mathbf{A}_2] &= [[a_{11}, a_{12}, a_{13}, a_{14}], [a_{21}, a_{22}, a_{23}, a_{24}, a_{25}]] \\ &= [[0, 1, 1, 0], [0, 1, 0, 1, 0]] \end{aligned}$$

The biases of each layer in the toy example are respectively a $1 \times 4$ vector $\mathbf{B_1}$, a $1 \times 5$ vector $\mathbf{B_2}$ and a $1 \times 2$ vector $\mathbf{B_3}$. Notice that, while the biases are involved in calculation for the network output $\mathbf{Y}$, they do not affect the input gradients. Thus we will first solve the weights for comparing the input gradients for flip pairs, and then finds the biases at last.

### C. Recovering the Last Layer Weights

In this section, we will detail the method to recover the values of the weights for last layer, illustrate the procedure on the toy example. As mentioned before, the main idea is that the difference between the input gradients of a flip pair provide the weights values. Thus we broke the method description into following parts: 1) *Activation Analysis* which describes how to retrieve the activation patterns from the side-channel measurements; 2) *Input Gradient Calculation* describes how to calculate the input gradient through an activation-inclusive pair; 3) *Flip Pair Searching* describes the search algorithm for a flip pair targeting a neuron on $i$-th layer; 4) *Weight Recovery* describes the recovery of weight values by comparing input gradients of a flip pair.

*1) Activation Analysis:* As described in Section III, we can use the side-channel measurement of the energy consumption to tell if each ReLU is activated. Table I displays a series of energy readings taken during querying. Each row represents the ReLU energy consumption of a specific ReLU neuron. With a threshold of $1348J$, neurons 2 and 3 have high energy consumption exceeding the threshold. Hence, we know that neurons 2 and 3 are active while the other three neurons are inactive,resulting in the activation vector $[0, 1, 1, 0, 0]$. Checking all ReLUs' energy consumption during the DNN execution this way, we get the activation pattern of the whole network.

TABLE I: Example of Energy Consumption of oneDNN ReLU in Intel SGX

| Index | Energy Consumption (J) | Binarized Energy $\mathbf{P}$ |
|---|---|---|
| 1 | 1333.94464111 | 0 |
| 2 | 1354.78417969 | 1 |
| 3 | 1355.43255615 | 1 |
| 4 | 1336.60076904 | 0 |
| 5 | 1333.91168213 | 0 |

**Algorithm 1:** Calculate Input Gradient

**Input :** $\mathcal{M}$-dimensional Input Vector $\mathbf{X} = [x_1, x_2, x_3, \ldots, x_m] \in$ Input dataset $\mathcal{S}$; Model $M$ contains $\mathcal{K}$ neurons with ReLU; Small positive perturbation $\delta$;

**Output:** The input gradient $\mathbf{I_k}$ for $x_k$ at input $\mathbf{X}$; $\mathcal{N}$-dimensional Output Vector $\mathbf{Y} = [y_1, y_2, y_3, \ldots, y_n]$;

Pick $x_k$ for adding perturbation
$\mathbf{X} = [x_1, x_2, x_3, \ldots, x_k, x_{k+1}, \ldots, x_m]$
$\mathbf{X}' = [x_1, x_2, x_3, \ldots, x'_k, x_{k+1}, \ldots, x_m]$ where $x'_k = x_k + \delta$
Query the model and record the corresponding activation vector set $\mathbf{A}$
Model output $\mathbf{Y}(\mathbf{X})$ with $\mathbf{A}$, $\mathbf{Y}'(\mathbf{X})$ with $\mathbf{A}'$
**if** $A = A'$ **then**
  $\mathbf{I} = (Y - Y')/\delta$
**end**
**else**
  $\delta = \delta/2$, then repeat from Step 3.
**end**

**Return:** The input gradient $\mathbf{I_k}$ for $x_k$ at $\mathbf{X}$

*2) Input Gradient Calculation:* For a network with $m$-dimensional input $\mathbf{X} = [x_1, \ldots, x_m]$ and $n$-dimensional output $\mathbf{Y} = [y_1, \ldots, y_n]$, we calculate the $k$-th input gradient $\mathbf{I}_k$ by Algorithm 1. We first query the model $M$ with the input $\mathbf{X}$ while also collect the vector of RAPL energy consumptions of all ReLU activations, denoted as $\mathbf{P}$. Then using the above Activation Analysis, we get the activation pattern $\mathbf{A}$ from $\mathbf{P}$. Next, we perturb the input on the entry $x_k$ by a small amount $\delta$ and check the activation pattern again with RAPL energy consumptions. For small enough perturbation in the input, the activation pattern remain the same, resulting in an activation-inclusive pair of $\mathbf{X}$ and $\mathbf{X}'$. The input gradient $\mathbf{I}_k$ is calculated through comparison of the outputs for the original input $\mathbf{X}$ and for the perturbed input $\mathbf{X}'$.

*3) Flip Pair Searching:* Here we describe the search method for a flip pair of inputs $\mathbf{X}$ and $\mathbf{X}'$ that have the same activation pattern except differing in the activation of a targeted neuron. The method is similar to part of the approach searching for witness of the target neuron [4], [5]. For searching the witness, using the side-channel measurements, we can start with any pair of inputs $\mathbf{X}$ and $\mathbf{X}'$ that differ in the activation of the target neuron. Then there is a witness on the line segment between $\mathbf{X}$ and $\mathbf{X}'$. To get more precision of the witness, binary search can be conducted to find a pair between $\mathbf{X}$ and $\mathbf{X}'$ that is much closer to each other but still differs in the target activation. When the pair of inputs are very close, only the target activation will flip but no more flip for activation of other neurons in the network. Then this pair is also a flip pair. For easy comparison of the input gradients in the weights recovery later, we also search for flip pairs that only differ in one input component $x_k$. We present the detailed steps in Algorithm 2.

**Algorithm 2:** Finding Flip Pair for A Specific Neuron

**Input :** $\mathcal{M}$-dimensional Input Vector $\mathbf{X} = [x_1, \ldots, x_m] \in$ Input dataset $\mathcal{S}$; Model $M$ contains $\mathcal{K}$ neurons with ReLU; Number $\sigma$;

**Output:** A Flip Pair $\mathbf{X}$ and $\mathbf{X}'$ at Neuron $N$

$\mathbf{X}' = [x_1, x_2, x_3, \ldots, x'_k, \ldots, x_m]$ where $x'_k = x_k + \sigma$
Query the model and record the corresponding activation pattern $\mathbf{A}$
Model output $\mathbf{Y}(\mathbf{X})$ with $\mathbf{A}$, $\mathbf{Y}'(\mathbf{X})$ with $\mathbf{A}'$
**if** *Bit flips in the target neuron* $N$ **then**
  **while** *Any other bit flips happened as well* **do**
    $\sigma = \sigma/2$ then binary searching for desired input.
  **end**
  **else**
    Record the activation pattern and input/output.
  **end**
**end**
**else**
  $\mathbf{X} = \mathbf{X}'$
  Increment $\sigma$ and repeat from Step 2.
**end**

**Return:** The input $\mathbf{X}$ and $\mathbf{X}'$, their activation patterns and outputs

*4) Recovering Weights :* Comparing the input gradients for a flip pair of the $j$-th neuron on the second to last layer reveals all last layer's weights connected to this neuron. We illustrate this on the toy example.

Say, we searched a flip pair $\mathbf{X}$ and $\mathbf{X}'$ for the last (i.e. the fifth) neuron on the second hidden layer, and then calculated their second input gradients $\mathbf{I}_2(\mathbf{X})$ and $\mathbf{I}_2(\mathbf{X}')$ respectively as described above. Assume that the activation pattern for input $\mathbf{X}$ is $\mathbf{A}(\mathbf{X}) = [[0, 1, 1, 0], [0, 1, 0, 1, 0]]$ as shown in Figure 4. Since the flip pair only differ in the activation of the 5-th neuron on the second hidden layer, $\mathbf{A}(\mathbf{X}') = [[0, 1, 1, 0], [0, 1, 0, 1, 1]]$. Since $\mathbf{A}_1(\mathbf{X}) = \mathbf{A}_1(\mathbf{X}')$ and $\mathbf{A}_2(\mathbf{X}') - \mathbf{A}_2(\mathbf{X}) = [0, 0, 0, 0, 1]$, using equation (3), the difference of the input gradients

$\mathbf{I}_2(\mathbf{X}') - \mathbf{I}_2(\mathbf{X})$ becomes

$$\begin{aligned} &= [\mathbf{W}_{1,2} * \mathbf{A}_1(\mathbf{X})][\mathbf{W}_2 * (\mathbf{A}_2(\mathbf{X}') - \mathbf{A}_2(\mathbf{X}))]\mathbf{W}_3 \\ &= [\mathbf{W}_{1,2} * \mathbf{A}_1(\mathbf{X})][0_4, 0_4, 0_4, 0_4, \mathbf{W}_2^{(5)}]\mathbf{W}_3 \\ &= [\mathbf{W}_{1,2} * \mathbf{A}_1(\mathbf{X})](\mathbf{W}_2^{(5)})\mathbf{W}_{3,5} \end{aligned}$$

where $\mathbf{W}_i^{(j)}$ denotes the $j$-th column of $\mathbf{W}_i$, and $0_4$ denotes a 4-dimensional column vector with all zero entries.

Thus the 5-th row $\mathbf{W}_{3,5}$ of the weight matrix $\mathbf{W}_3$ is known from $\mathbf{I}_2(\mathbf{X}') - \mathbf{I}_2(\mathbf{X})$ except for an unknown constant $[\mathbf{W}_{1,2} * \mathbf{A}_1(\mathbf{X})](\mathbf{W}_2^{(5)})$. The sign of $\mathbf{I}_2(\mathbf{X}') - \mathbf{I}_2(\mathbf{X})$ can also be discerned. The difference in input gradients $\mathbf{I}_k(\mathbf{X}') - \mathbf{I}_k(\mathbf{X})$ provides insights into how the input gradients adjust due to variations in the input $\mathbf{X}$. As demonstrated in Algorithm 2, by modifying the input entry $x_k$, we can alter the activation pattern. We thus simplify it to $\mathbf{I}_k(x'_k) - \mathbf{I}_k(x_k)$. This solely depends on the weights and activation patterns of each layer. If we retain the same activation pattern, the relationship between the difference in outputs $\Delta\mathbf{Y} = \mathbf{Y}(x'_k) - \mathbf{Y}(x_k)$ and the difference in inputs $\sigma = (x'_k - x_k)$ will form a linear relationship. For the output $\mathbf{Y}$, the change can be represented as:

$$\begin{aligned} \mathbf{Y}(x'_k) - \mathbf{Y}(x_k) &= (\mathbf{I_k}(x'_k) - \mathbf{I_k}(x_k))(x'_k - x_k) + \mathbf{Z} \\ \Delta\mathbf{Y} &= (\mathbf{I_k}(x'_k) - \mathbf{I_k}(x_k))\sigma + \mathbf{Z} \end{aligned}$$

Here, $\Delta\mathbf{Y}$ represents the difference in outputs, $\sigma$ represents the difference in inputs, and $\mathbf{Z}$ is the constant. When $k = 2$, the term $\mathbf{I}_2(x'_2) - \mathbf{I}_2(x_2)$ represents the previously obtained difference, $\mathbf{I}_2(\mathbf{X}') - \mathbf{I}_2(\mathbf{X})$, and we need to determine its sign. If we keep the activation pattern constant and observe an increase in the entries of $\Delta\mathbf{Y}$ when $\sigma$ increases, then the corresponding entries of $\mathbf{I}_2(\mathbf{X}') - \mathbf{I}_2(\mathbf{X})$ are positive. Conversely, if the entires of $\Delta\mathbf{Y}$ decrease, then the corresponding entries of $\mathbf{I}_2(\mathbf{X}') - \mathbf{I}_2(\mathbf{X})$ are negative.

At this point, we have successfully determined the weights $\mathbf{W_{3,5}}$ up to an unknown positive multiplicative constant. By repeating this procedure for all neurons in the last hidden layer, we can derive all weights in that layer.

### D. Recovering Weights of Earlier Layers

To recover the weights in earlier layers, we first generalize the definition of the input gradients to $i$-th layer: the derivative of the $i$-th layer output with respect to the $k$-th input component, denoted as $\mathbf{I}_k^{(i)}$. Thus, for a network with depth $D$, the $\mathbf{I}_k$ defined before is just $\mathbf{I}_k^{(D)}$. Similar to (3), for layers except the last, we have

$$\mathbf{I}_k^{(i)} = (\mathbf{W}_{1,k} * \mathbf{A}_1) \cdot (\mathbf{W}_2 * \mathbf{A}_2) \cdots (\mathbf{W}_i * \mathbf{A}_i). \quad (4)$$

With the $D$-the layer weights $\mathbf{W}_D$ recovered, we can repeat the above procedure to recover the weights of the $(D-1)$-th layer. We search for a flip pair $\mathbf{X}$ and $\mathbf{X}'$ for the $j$-th neuron on the $(D-2)$-th layer. Using the fact that $\mathbf{I}_k = \mathbf{I}_k^{(D)} = \mathbf{I}_k^{(D-1)}\mathbf{W}_D$, we can also calculate $\mathbf{I}_k^{(D-1)}(\mathbf{X})$ and $\mathbf{I}_k^{(D-1)}(\mathbf{X}')$, from $\mathbf{I}_k^{(D)}(\mathbf{X})$, $\mathbf{I}_k^{(D)}(\mathbf{X}')$ and $\mathbf{W}_D$. Then the difference $\mathbf{I}_k^{(D-1)}(\mathbf{X}) - \mathbf{I}_k^{(D-1)}(\mathbf{X}')$ provides the weights $\mathbf{W}_{D-1,j}$. Iteratively repeat the process, we can recovers the weights layer by layer until all weights are recovered.

Note that most CNNs are constructed from several convolutional blocks followed by a number of fully-connected layers. These fully-connected layers can be treated as MLPs, and our input gradient-based model extraction algorithm can be directly applied to them. The convolutional blocks, which consist of several convolutional layers with ReLU activation and pooling layers, is challenging to determine which entries are selected by the pooling layer. We adopt the similar method presented in [5]. This method involves solving for weights from the first layer to the last, which does not require prior knowledge of the pooling layer patterns from earlier layers. Thus, we continue to identify the critical points from our flip pairs using a binary search while monitoring the RAPL.

### E. Recovering Biases

To obtain the biases for each layer, we construct equations based on the input and output of flip pairs, as opposed to input gradients, given that the input gradients do not account for biases in their computations. Contrary to how we recover weights - from the last layer to the first - biases are recovered starting from the first layer, progressing towards the last. This method allows us to isolate and eliminate the influence of biases from subsequent layers during the extraction process.

We denote $b_{ij}$ as the bias setting to the $j$-th neuron in the $i$-th layer. **Comparing the outputs for a flip pair of the $j$-th neuron in the $i$-th layer reveals the bias $b_{ij}$**. From equation (1), it is evident that the bias $b_{ij}$ is influenced only by the weights $\mathbf{W_k}$ and activation pattern $\mathbf{A_k}(\mathbf{X})$ where $k > i$. This implies that, in the input-output equation, **the bias is impacted solely by the weights and activation patterns of subsequent layers**. This presents us with the opportunity to calculate biases from the first layer up to the last layer. If we are recovering the bias in Layer $L_i$, we can maintain the same activation pattern in the subsequent layers $L_{i+1}, L_{i+2}, \ldots, L_D$.

We use the toy example illustrated in Figure 4. The four layers with heights $(h_0, h_1, h_2, h_3) = (3, 4, 5, 2)$. So the biases of each layer are a $1 \times 4$ vector $\mathbf{B_1}$, a $1 \times 5$ vector $\mathbf{B_2}$ and a $1 \times 2$ vector $\mathbf{B_3}$. Once we have successfully recovered all the weight matrices $\mathbf{W_1}$, $\mathbf{W_2}$, and $\mathbf{W_3}$, we can use pairs of inputs and outputs to solve the biases $\mathbf{B} = [\mathbf{B_1}, \mathbf{B_2}, \mathbf{B_3}] = [[b_{11}, b_{12}, b_{13}, b_{14}], [b_{21}, b_{22}, b_{23}, b_{24}, b_{25}], [b_{31}, b_{32}]]$. A flip pair $\mathbf{X}$ and $\mathbf{X}'$ for the last (i.e. the forth) neuron on the first hidden layer, and calculated their first output $y_1$ and $y'_1$. Assume that the activation pattern for the input $\mathbf{X}$ is $\mathbf{A}(\mathbf{X}) = [[0, 1, 1, 0], [0, 1, 0, 1, 0]]$. The flip pair only differ in the activtion of the 4-th neuron on the first hidden layer, $\mathbf{A}(\mathbf{X}') = [[0, 1, 1, 1], [0, 1, 0, 1, 0]]$. Since $\mathbf{A}_1(\mathbf{X}') - \mathbf{A}_1(\mathbf{X}) = [0, 0, 0, 1]$ and $\mathbf{A}_2(\mathbf{X}) = \mathbf{A}_2(\mathbf{X}')$, using equation (1), the difference of the output $\mathbf{Y}(\mathbf{X}') - \mathbf{Y}(\mathbf{X})$

$$\begin{aligned} = \quad & [\mathbf{X}'[\mathbf{W_1} * \mathbf{A_1}(\mathbf{X}')] - \mathbf{X}[\mathbf{W_1} * \mathbf{A_1}(\mathbf{X})]] \\ & [\mathbf{W_2} * \mathbf{A_2}(\mathbf{X})](\mathbf{W_3}) \\ & +((\mathbf{A}_1(\mathbf{X}') - \mathbf{A}_1(\mathbf{X})) * \mathbf{B_1})(\mathbf{W_2} * \mathbf{A}_2(\mathbf{X}))(\mathbf{W_3}) \end{aligned}$$

From the given formula, $\mathbf{B_1}$ appears to be the **only unknown**. Let's assume $\mathbf{C}(\mathbf{X}, \mathbf{X}', \mathbf{W}) = [\mathbf{X}'[\mathbf{W_1} * \mathbf{A_1}(\mathbf{X}')] - \mathbf{X}[\mathbf{W_1} * \mathbf{A_1}(\mathbf{X})]][\mathbf{W_2} * \mathbf{A_2}(\mathbf{X})](\mathbf{W_3})$, where $\mathbf{C}(\mathbf{X}, \mathbf{X}', \mathbf{W})$ is a function that depends only on the known values $\mathbf{X}$, $\mathbf{X}'$, and $\mathbf{W}$. Thus, we can express $y'_1 - y_1$

$$= \quad \mathbf{C}(\mathbf{X}, \mathbf{X}', \mathbf{W}) + (w_{242}w_{321} + w_{244}w_{341})b_{14}$$

With the known values of $\mathbf{X}$, $\mathbf{X}'$, $\mathbf{W}$ and $\mathbf{Y}$, we can determine $\mathbf{C}(\mathbf{X}, \mathbf{X}', \mathbf{W})$, $[w_{242}, w_{321}, w_{244}, w_{341}]$, $y'_1$, and $y_1$. This information allows us to calculate $b_{14}$. By iterating over the flip pairs of neurons in the first hidden layer, we can recover the biases $B_1 = [b_{11}, b_{12}, b_{13}, b_{14}]$.

To recover all biases, we continue this process for each neuron in the second layer and finally, the last layer. As such, we are able to recover the complete bias matrix, $\mathbf{B} = [[b_{11}, b_{12}, b_{13}, b_{14}], [b_{21}, b_{22}, b_{23}, b_{24}, b_{25}], [b_{31}, b_{32}]]$.

For the retrieval of biases in convolutional neural networks, our proposed methodology can be effectively applied to both the convolutional layers and the fully-connected layers. This is feasible because there are no obstacles introduced by the pooling layer.

## V. Evaluation

In this section, we evaluate the efficacy of our RAPL-based model extraction strategy. Our RAPL-based model extraction leverages a software-based power side channel to gather critical data about DNN model execution, specifically the activation patterns. We integrate this activation patterns information through an innovative model extraction algorithm based on input gradients. This reduces the complexity of model extraction significantly compared to previous cryptanalytic model extraction algorithms. As a result, our purely software-based attack can extract significantly more complex and larger models than prior model extraction work. A potential countermeasure of the algorithm is neuron shuffling during the runtime execution of deep learning models. If the ReLU neurons execution order is randomized at the runtime, this can prevent identification of which neuron a RAPL measurement corresponds to, thus not allow the correct determination of the activation pattern.

Compared to prior work [5], our weight recovery algorithm significantly reduce the needed query complexity by utilizing the activation patterns. Searching the flip pairs is easier than their searching critical points, the side-channel information allows us to focus the search on a target neuron. The prior work using only input/output of the query can not directly decide which neuron the critical point corresponds to. The uncertainty about the correspondence requires much more queries to find a critical point corresponding to the target neuron.

We compare both methods on their complexity of recovering all weights in a network with a total of $N$ hidden neurons. [5] first find many witnesses to critical points but not knowing which neuron these critical points are for. Then to choose critical points to a target neuron, they used trial and error method by assuming each critical point did correspond to the target neuron. Consistent solutions are arrived when two critical points do correspond to the same target neuron, while the solutions differ when not both critical points correspond to the same neuron. For this method to work, they need to first find a candidate set of critical points with size $S$, where $S >> N \log N$, to ensure with high probability that each neuron has multiple corresponding critical points in the candidate set. In comparison, our search can be pinpointed to the target neuron through the RAPL measurement and only $N$ flip pairs are needed. Furthermore, when the depth of network grows, the cryptanalytic method of [5] require exponentially more precision in the search for initial layer critical points. Since their critical points calculation depend on the extracted weights from previous layers, any errors in those extracted weights propagate to the next layer and compound as the analysis progresses into deeper layers. In contrast, our flip pairs search can be pinpointed to the target neuron with RAPL information specific to the neuron without any precision degradation as we progress through the layers. Thus the complexity reduction of our method compared to [5] is more significant for a deeper networks.

Table II presents the number of queries needed to recover weights of a whole network by our RAPL-based method versus the prior cryptanalytic model extraction work.

TABLE II: Efficacy of our extraction attack with prior works

| Architecture | Number of Parameters | Prior works' Queries | Our Queries |
|---|---|---|---|
| 784-32-1 | 25,120 | $2^{18.2}$ [4], $2^{19.2}$ [5] | $2^{10.7}$ to $2^{14.9}$ |
| 784-128-1 | 100,480 | $2^{20.2}$ [4], $2^{21.5}$ [5] | $2^{11.8}$ to $2^{15.1}$ |
| 10-10-10-1 | 210 | $2^{22}$ [27], $2^{16}$ [5] | $2^{6.9}$ to $2^{9.2}$ |
| 10-20-20-1 | 420 | $2^{25}$ [27], $2^{17.1}$ [5] | $2^{7.6}$ to $2^{10.1}$ |
| 40-20-10-10-1 | 1,110 | $2^{17.8}$ [5] | $2^{8.3}$ to $2^{11}$ |
| 80-40-20-1 | 4,020 | $2^{18.5}$ [5] | $2^{9.1}$ to $2^{11.9}$ |
| Lenet-5 | 44,426 | $---$ | $2^{13.4}$ to $2^{16.9}$ |

Our method, utilizing the RAPL information, reduces the number of needed queries in every networks studied. The improvement becomes more pronounced for deeper networks. To retrieve a 5-layer MLP, we need $2^{11}$ queries, which is only $0.8\%$ of $2^{17.8}$ in prior work. Our method is also able to recover weights of a commonly used CNN, Lenet-5 [28] trained on MNIST dataset [29]. We achieve 100% fidelity on test set (contains 10,000 samples) in $2^{16.9}$ queries.

## VI. Conclusion

In this paper, we introduce a novel approach to extract DNN models leveraging the Intel RAPL technique, a software-based power side-channel. The RAPL power side-channel can reveal the direction of ReLU activations, enabling us to discern the activation patterns and subsequently identify flip pairs for neurons. Our input gradient-based model extraction algorithm utilizes the activation pattern of these flip pairs in conjunction with input-output pairs to recover all parameters in the DNN model. This powerful combination of side-channel and algorithmic approaches allows us to extract a 5-layer MLP using only 0.8% of the queries required by previous methods.

The RAPL-based power side-channel presents an effective means to decipher piece-wise activation functions. While we've specifically discussed ReLU power leakage, this method is applicable to any vulnerable piece-wise activation function implementation in DNN frameworks. The RAPL-based power side-channel is also compatible with ARM processors, which include RAPL sensors. With an appropriate power magnification strategy, we anticipate being able to adapt our method to the ARM platform. Our model extraction algorithm is not limited to our side-channel, but rather has broad applicability to any method capable of extracting the activation pattern of the model and recognizing the inputs and outputs. We are optimistic about the potential for more efficient methods (or side-channels) to leak activation patterns in the future.

## REFERENCES


[1] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[2] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," 2017.

[3] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, Ú. Erlingsson, A. Oprea, and C. Raffel, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2633–2650. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting

[4] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[5] N. Carlini, M. Jagielski, and I. Mironov, "Cryptanalytic extraction of neural network models," in *Advances in Cryptology – CRYPTO 2020*, D. Micciancio and T. Ristenpart, Eds. Cham: Springer International Publishing, 2020, pp. 189–218.

[6] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 601–618.

[7] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "Deepsniffer: A dnn model extraction framework based on learning architectural hints," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 385–399. [Online]. Available: https://doi.org/10.1145/3373376.3378460

[8] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 515–532.

[9] C. Gongye, Y. Fei, and T. Wahl, "Reverse-engineering deep neural networks using floating-point timing side-channels," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[10] M. Randolph and W. Diehl, "Power side-channel attack analysis: A review of 20 years of study for the layman," *Cryptography*, vol. 4, no. 2, p. 15, 2020.

[11] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, mar 2018. [Online]. Available: https://doi.org/10.1145/3177754

[12] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[13] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3152701.3152706

[14] I. Corporation, *Intel Software Guard Extensions Programming Reference*.

[15] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[18] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[19] "Intel oneapi deep neural network library," https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html.

[20] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[22] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt, "Model reconstruction from model explanations," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*, ser. FAT* '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–9. [Online]. Available: https://doi.org/10.1145/3287560.3287562

[23] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2015.

[24] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," 2017.

[25] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," 2014.

[26] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: removing noise by adding noise," 2017.

[27] D. Rolnick and K. Kording, "Reverse-engineering deep ReLU networks," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 8178–8187. [Online]. Available: https://proceedings.mlr.press/v119/rolnick20a.html

[28] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, no. 5, p. 14, 2015.

[29] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.