Reverse-Engineering Deep Neural Networks Using Floating-Point Timing Side-Channels

Cheng Gongye, Yunsi Fei, Thomas Wahl
Northeastern University, Boston, MA, US
gongye.c@husky.neu.edu, yfei@ece.neu.edu, t.wahl@northeastern.edu

***Abstract*—Trained Deep Neural Network (DNN) models have become valuable intellectual property. A new attack surface has emerged for DNNs: model reverse engineering. Several recent attempts have utilized various common side channels. However, recovering DNN parameters, weights and biases, remains a challenge. In this paper, we present a novel attack that utilizes a floating-point timing side channel to reverse-engineer parameters of multi-layer perceptron (MLP) models in software implementation, entirely and precisely. To the best of our knowledge, this is the first work that leverages a floating-point timing side-channel for effective DNN model recovery.**

***Index Terms*—Deep learning, floating-point arithmetic, multilayer perceptrons (MLP), reverse engineering, side-channel attacks**

I. Introduction

Deep learning (DL) has become a foundational means for solving grand societal challenges, disrupting many application domains with superior performance. Trained Deep Neural Networks (DNNs) models have become a commodity and widely deployed in the cloud as services or adopted in edge devices. These trained models are valuable intellectual properties for the following reasons. Training DNN models for applications requires access to high-quality, often proprietary datasets and also demands a considerable amount of computational resources (e.g., using high-end GPUs [1]). Typically it also requires machine learning experts and domain experts to work together towards selecting network structures suitable for the task at hand, pre-processing the dataset, and fine-tuning the model structure and hyper parameters.

Given the high value of today's DNN models, an adversary has a strong incentive to reverse-engineer a trained DNN model and recover a near-identical one. Knowing the DNN model also facilitates other attacks, including membership inference attacks [2], attacks with adversarial examples that can look normal to the human eye but yield a wrong output [3], and active fault injection attacks that maliciously modify the model parameters to disrupt the services [4].

However, "stealing" model parameters through software attacks is hard. Models can be encrypted to protect the confidentiality of the parameters. They can also be encapsulated in trusted environments, e.g., Intel SGX or ARM TrustZone, and the end-user of the model can only access it by providing the input and receiving the result, i.e., in a black-box fashion. Recently, there have been several attempts to reverse-engineer DNN models using side-channel attacks, leveraging different types of side channels, with different attack targets (structure, hyper parameters, parameters, or inputs), and running on different implementation platforms (hardware vs. software). However, they all have limitations; we discuss some in the following.

This work was supported in part by the National Science Foundation under grants SaTC-1929300 and SaTC-1563697.

A. Related Work

Hua et al. [5] targeted a DNN accelerator on FPGA and reduced the search space of DNN model architectures (both structure characteristics and hyper parameters) by observing off-chip memory access patterns and timing. This attack has the potential to recover weights when the dynamic zero pruning is used. Cache side-channel attacks have been exploited for stealing DNN models in software implementations. Yan et al. [6] reduced the search space of DNN model architectures, by tracking the usage of tiled general matrix multiplication (GEMM) in DNN inference. Hong et al. [34] recovered the DNN structure by observing function invocations through the Flush+Reload technique. However, their work does not tackle hyper-parameters. Other side channels, including power, EM, and timing, are exploited to recover the DNN structure and weights [7]. However, they can only recover the weights to low accuracy, e.g., before "4^{th} place after the decimal point". They did not address bias recovery. Dong et al. [8] developed a floating-point timing attack to recover the input images of a DNN implemented on microcontrollers. Their attack targets the input images rather than the DNN model; their timing model is a shorter timing for multiplication with a zero input, which is only accurate for microcontrollers but not for general-purpose processors or GPUs.

B. Contributions and Organization

None of the previous work was able to recover all the parameters of DNN models accurately. In this work, we utilize the floating-point timing side channel to recover all parameters (weights and biases) of DNN models executing in software entirely. To the best of our knowledge, this is the first work that leverages a floating-point timing side channel in DNN model recovery. We also show the proposed attack applies to different DNN model types.

The rest of the paper is organized as follows. Section II-C introduces some background for our reverse-engineer attack. Section III describes the procedures and details of the attack.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

We use a case study in Section IV to evaluate the attack performance. In Section V, we discuss the attack generality and extendability. We conclude this work in Section VI.

II. Background

In this section, we present some background on floating-point operations, the basic DNN model (multi-layer perceptron, MLP), and our threat model.

A. Floating-Point Timing Side-Channel

IEEE-754 is the most widely adopted floating-point (FP) number standard. For an IEEE-754 single-precision (SP) floating-point number, the 32-bit encoding is composed of a single-bit *sign* (S), an 8-bit *exponent* (E), and a 23-bit *mantissa* (M). We target this format in this work, but the attack is otherwise agnostic to the FP format.

Depending on the values of the exponent and mantissa, there are five types of FP numbers, as shown in Table I.

TABLE I
IEEE Single Precision Floating Point Number Types

Type	Exponent	Mantissa	Formula
normal	[1, 254]	any	$(-1)^S \cdot 2^{E-127} \cdot 1.M$
zero	zero	zero	0
subnormal	zero	non-zero	$(-1)^S \cdot 2^{E-126} \cdot 0.M$
Infinities	255	zero	Inf
Not-a-Number	255	non-zero	NaN

A subnormal floating-point number represents values smaller than that of normal floating-point numbers. It enables the correct representation of the result of floating-point operations that underflow. On commercial CPUs, there are typically dedicated floating-point arithmetic units and registers for normal floating-point operations, in addition to the integer operators in the datapath. However, since subnormal floating-point numbers are less frequent, there is no dedicated hardware support for them on modern processors. Instead, processors may have hardware to detect subnormal operands (whether they are sources or destination) and implement operations on them in software (i.e., dispatch them onto microcode executions), which makes some of them much slower than normal operations [9].

A longer execution time potentially leaks information about the nature of the operands—opening up a floating-point timing side channel. Andrysco et al. exploit this side channel to steal pixels in a browser [10].

B. Deep Neural Networks

DNN models can be viewed as cascaded connections of multiple functional layers, such as fully-connected (FC) layer, convolutional (CONV) layer, and pooling (POOL) layer, to extract features for classification or detection. A very common DNN architecture widely used in classification applications is the multi-layer perceptron (MLP), depicted in Fig. 1. A MLP consists of multiple FC layers of neurons and features a feed-forward network that maps input features into outputs. Each layer is fully connected to the previous and the subsequent layer, taking a feature vector from the previous layer, processing it, and generating an output feature vector for the next layer. The entire layer computation can be viewed as a multiplication of a weight matrix with the input vector, followed by the bias vector addition and activation function application. For example, a linear algebra representation of the first hidden layer is:

$$\mathbf{l_1} = Activation\left(\mathbf{W_1} \cdot \mathbf{l_0} + \mathbf{b_1}\right) \tag{1}$$

where $\mathbf{l_0}$ is the input feature vector of length of m, $\mathbf{W_1}$ is the first-layer weight matrix of dimension $n \times m$, where n is the number of neurons, $\mathbf{b_1}$ is the bias vector of length n, and $\mathbf{l_1}$ is the output feature vector of length n. For the example in Fig. 1, we have $n = 4$ and $m = 6$. Note that each neuron in the first hidden layer applies one of the row vectors of the weight matrix to the input vector to generate one item for the output feature vector. For the second hidden layer, the weight matrix is of size 3×4.

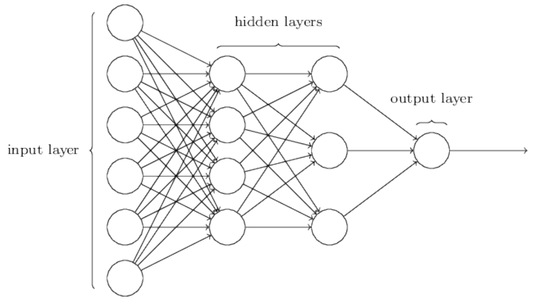

Fig. 1. An example multi-layer perceptron

C. Threat Model

Our threat model is a black-box attack targeting DNN inference. The trained MLP model is either encrypted or protected in a trusted execution environment. Therefore, the adversary has no knowledge of the MLP parameters. However, he can query the model with arbitrary input and can observe the output in the last layer. He also knows the model structure in terms of the number of layers and the number of neurons in each layer, either from commonly available datasets or from a visualization of the DNN inference through simple power analysis. He can measure the timing of each layer of operations with high precision. Timing measurements can be achieved by either analyzing the cache access pattern [6] or through visual inspection of power traces [7].

III. The Reverse-Engineering Attack

In this section, we describe our method of reverse-engineering the weights and biases of an MLP based on floating-point timing side-channel leakage. The model is recovered layer by layer. The organization of this section is as fellows. We start with the low-level timing models for floating-point operations on an x86 processor. Then we reverse-engineer the weights and biases in the first hidden layer. Attacking the remaining layers is very similar to attacking the first layer; we point out the minor differences.

A. Timing Models of x86 Floating-Point Operations

Inference of an MLP is essentially a series of multiplication and addition operations. Hence, for this work, we only need to consider abnormal timing of these floating-point operations. We define abnormal timing as the consumption of noticeably more CPU cycles by an executing floating-point operation than by a normal floating-point operation.

1) FP Multiplication Timing Model: Consider a floating-point multiplication where a, b and c are non-zero: $a \cdot b = c$. For most cases, if one of a, b, or c is a subnormal floating-point number, this operation will feature abnormal timing (take much longer). However, if either operand or the result is zero, we will not observe abnormal timing. We develop a suite of microbenchmarks to characterize the timing model of x86 floating-point multiplications, shown in Table II. All experiments are performed on a workstation with Intel i7-7700 quad-core processor and 2×8GB Dual-channel DDR4 memory. We find an average extra timing of 114 cycles for abnormal operations, which we denote as σ.

TABLE II
TIMING MODEL FOR FLOATING-POINT MUTIPLICATIONS

Case	Operation	CPU cycles
1	$normal \cdot normal = normal$	10
2	$normal \cdot normal = subnormal$	124
3	$subnormal \cdot normal = normal$	124
4	$subnormal \cdot normal = subnormal$	124
5	$subnormal \cdot subnormal = 0$	10
6	$subnormal \cdot 0 = 0$	10

2) FP Addition Timing Model: For a floating-point addition, $a + b = c$, this operation will have an abnormal timing when $|c| \in (1\text{e}{-43}, \max_{sn})$ and $|a| \in (\min_n, 6\text{e}{-33})$ (from our observations on the experiment platform), where $\max_{sn}$ is the largest (single-precision) subnormal number ($\approx 1.1754942\text{e}{-38}$) and $\min_n$ is the smallest normal number ($\approx 1.1754944\text{e}{-38}$). Similarly, we run microbenchmarks to characterize the timing model of FP addition and find that the σ is also about 114 cycles.

Previous work that utilizes floating-point timing side channels mainly focuses on multiplications and divisions. In this work, we take advantage of the timing leakage of additions too (which are frequent in DNN inference). In the following subsections, we will show how we leverage these two timing models to reverse-engineer the weights and biases.

B. Recovering the First Layer

The linear algebra representation of the first layer is shown in Equation (1). Our goal of attacking the first layer is to recover all the elements of $\mathbf{W_1}$ and $\mathbf{b_1}$, by only varying $\mathbf{l_0}$ and observing the timing. In this paper, we assume the activation function to be a rectified linear unit (ReLU), one of the most effective and widely adopted activation functions.

Our approach proceeds in three steps: 1) recover the absolute values of each column of the weight matrix; 2) arrange the weights to figure out weights belonging to the same row and find their relative signs; and 3) recover the bias vector and the actual signs of all parameters in the first layer.

1) Column Absolute Values: This attack utilizes the first timing model presented in III-A1. We utilize case 2 in Table II, where the product is subnormal, and the inputs are normal numbers within the range of $[min_n, 1]$. As floating-point numbers are signed values, the signs are processed separately and do not affect the multiplication timing. We first ignore the signs and only recover the absolute values. For an FP multiplication with two operands, assuming one operand is unknown but fixed (e.g., a weight x), we can control and vary the other operand (e.g., an input i). By observing the operation time under different inputs and utilizing the timing model for multiplications, we can recover the unknown operand with high precision.

Suppose that, for two inputs a and a' satisfying $a > a'$, the multiplication results in significantly different timing: $T(a \cdot x) < \sigma < T(a' \cdot x)$. Then we conclude that the first multiplication produces a normal output while the second produces a subnormal output. Then the unknown value x must satisfy $\frac{\min_n}{a} \leq x \leq \frac{\max_{sn}}{a'}$. We gradually reduce the gap between a and a' until the distance between $\frac{\min_n}{a}$ and $\frac{\max_{sn}}{a'}$ is less than our choice of precision, ϵ. Then our guess of the unknown operand converges to $x \approx \frac{\min_n}{a}$. Values a and a' can be controlled using binary search in the range $[min_n, 1]$.

For the first DNN layer, each neuron performs a vector multiplication of a weight row and the input vector. In software implementation without parallelism, these neuron computations are carried out in a sequence, and all contribute to the total timing. The finest-grained timing observation we assume an adversary has is the execution time of the first layer, rather than individual neurons. By setting the input vector to have only one non-zero value, each neuron's operation is reduced to one multiplication and one addition. For example, to focus on the first column, we set $l_0[1] = a$, $l_0[2 : m] = 0$ where $\mathbf{l_0}$ is the input vector of length m. With n neurons, the observed first-layer computation time is the sum of the times for n multiplications with fixed other value. The timing model for the first layer becomes:

$$T_{layer1}(a) = \sum_{i=1}^{n} T(a \cdot \mathbf{W}_1[i, 1]) + T_{others}$$

where $T(a \cdot \mathbf{W}_1[i, 1])$ is the computation time of each neuron, and T_{others} captures the execution time of other computations such as addition and activation function, which is assumed to be a constant. If all the n multiplications produce normal results, the total time is minimized: c (e.g., when a = 1), while if all produce subnormal results, the total time is maximized: $c + n \cdot \sigma$ (e.g., when $a = min_n$). As the input a varies, the total execution time varies within $[c, c + n \cdot \sigma]$.

Our attack consists of two steps. First, find a vector $\mathbf{A} = \{a_1, a_2, ...a_n\}$ with the n values in decreasing order, such that $T_{layer1}(a_i) = c + i \cdot \sigma$. That is, as the value a decreases, more and more subnormal products are generated by the first layer. The second step is to find the n weight values.

We envision that in the range $[min_n, 1]$, there exist n such values, namely $A_0[i] = max_{sn}/V[i], i \in [1, n]$, where $V[i]$

are the n weight values. We treat these n values as reference points, which divide the range of $[min_n, 1]$ into $n+1$ segments for the value of a, with the $T_{layer1}(a)$ for each segment decreases from $c + n \cdot \sigma$ to c, from the left most to the right. We are finding a vector $\mathbf{A}$ such that its n values partition the range of a into $n + 1$ intervals, where each of the intervals contains one such reference value $A_0[i]$. If we can trace this value with the interval known, we can recover the weight.

The trivial way to find $\mathbf{A}$ vector is to scan the value of a from large to small with a well-controlled stride, which incurs significant computation delay. To improve the speed of the attack, we employ a recursive binary search method to find $\mathbf{A}$, as illustrated in Fig. 2.

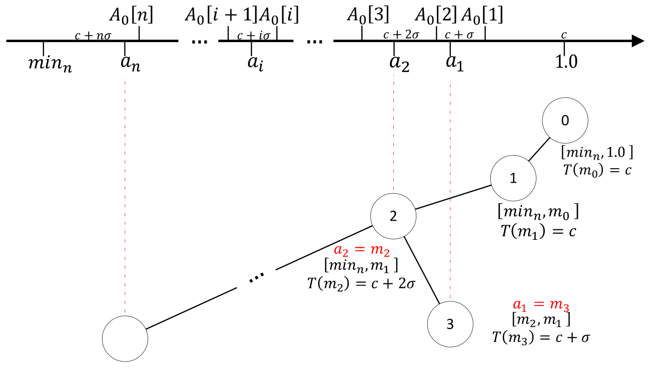

Fig. 2. The binary search method to find the A vector

We start from the range $[min_n, 1]$ for a (as the root node for a tree), and mark the two ends as L and R with $T_{layer1}(L) = c + n\sigma$ and $T_{layer1}(R) = c$. We first set $m_0 = (L + R)/2$ the middle point, and evaluate $T_{layer1}(m_0)$. If $T_{layer1}(L) - T_{layer1}(m_0) > \sigma$ and $T_{layer1}(m_0) - T_{layer1}(R) > \sigma$, we keep both the two half ranges $[L, m_0]$ and $[m_0, R]$ (as two children nodes of the root node and keep exploring them). When $T_{layer1}(m_0) = c + i\sigma$, we get $a_i = m_0$. However, if $T_{layer1}(L) - T_{layer1}(m_0) \leq \sigma$, we will not explore the left half range, i.e., no such left child. Similarly for $T_{layer1}(R)$. We keep growing the binary tree by splitting a range represented by a parent node into two halves, and use the prior criterion to decide whether to generate any child. In the graph, each node represents a range, marked by three T values, $T_{layer1}(L)$, $T_{layer1}(R)$, and $T_{layer1}(m)$. The binary tree will stop until we have found all the n a_i values (parent nodes in the graph).

After obtaining $\mathbf{A}$, we have all the intervals. Each of the intervals contains one of the reference points. We adopt a binary search method again to find the reference point x (approximate), falling into the user-defined precision ϵ. We will recover the weight by max_{sn}/x. We demonstrate this method in Algorithm 1; it has a complexity of $\mathcal{O}(\log n)$. Other columns are attacked by changing the location of the non-zero value in the input.

2) Weights with Relative Signs in Each Row: After the first step, we have recovered all the weights in each column of $\mathbf{W_1}$. However, we do not know the order of the weights, i.e., which row each weight belongs to. We represent these columns of weights as m vectors of length n: $\mathbf{V_1}, \mathbf{V_2}, \ldots, \mathbf{V_n}$. In this subsection, we present the method and algorithms to identify the weights of the same row.

Algorithm 1: Algorithm for finding the reference point in each interval (T represents T_{layer1})

```
Input  : $\epsilon$, $[l, r]$ (the interval containing the target)
Output: $x$ (the target value)
$t \leftarrow T(r)$
while $r - l > \epsilon$ do
    $mi$ = $(l + r)/2$
    if $T(mi) - t \geq \sigma$ then
        $l \leftarrow mi$
    else
        $r \leftarrow mi$
return $x = (l + r)/2$
```

Recovering all the locations of weights together is hard because we can only control the input and observe the timing. Analyzing the entire weight matrix by evaluating all the combinations of weights is possible but may be computationally prohibitive. We adopt an iterative technique to accomplish this task. To create a reference point for each row, we pick the first column of the weight matrix and sort its values. Then, for each element of the first column, we identify which element in each of the remaining columns belongs to the same row, i.e., we recover a weight row vector. We repeat this step for all elements in the first weight column and recover all the n weight rows. Hence, we have reduced the original problem of identifying if two elements from two columns are in the same row.

In this attack, we employ the timing model of floating-point addition, described in III-A2. For an addition operation, we fix one operand to be constant x. We know that the other operand is an element from a vector which is the product of a known vector and a variable a, $a \cdot [V[1], V[2], V[n]]^T$ where the values $V[i]$ are all positive absolute values and sorted in an increasing order. We first let $a = \Delta/V[1]$ such that $x - \Delta$ will trigger a much longer execution. We then set $a = \Delta/V_1$ and $a = -\Delta/V_1$. If we do not observe different execution times, it indicates $V[1]$ is not the element in the addition with x. We repeat this step for all other elements of the vector until we observe different execution times.

To apply this algorithm to our problem of finding weights from two columns on the same row, we can set all but the two corresponding items in the input feature to be non-zero. For example, to determine which weight in second column is in the same row as the first element of the input column, we set $\mathbf{l_0}[1] = a$, $\mathbf{l_0}[2] = b$, both non-zero normal numbers, and $\mathbf{l_0}[3:m] = 0$. Then the timing model of the first layer is:

$$T_{layer1}(a, b) = \sum_{i=1}^{n} T(a \cdot \mathbf{W_1}[i, 1] + b \cdot \mathbf{W_1}[i, 2]) + T_{others}$$

where T_{others} captures the execution times for other operations (multiplication, bias addition, etc.) and can be assumed to be a deterministic constant. Let $a = x/\mathbf{V}_1[1]$, $b = \Delta/\mathbf{V}_2[1]$, such that $T(x+\Delta) < \sigma \leq T(x-\Delta)$. Then we measure the execution time $t_1 = T_{layer1}(a, b)$, $t_2 = T_{layer1}(a, -b)$. If $t_1 \neq t_2$ with the difference at the level of σ, the timing difference must come from a subnormal addition. In this situation, we know that $\mathbf{V}_2[1]$ is in the same row with $\mathbf{V}_1[1]$. Otherwise, it is not and we need to move on to other elements of Column 2. We keep a unchanged and set $b = \Delta/\mathbf{V}_2[j]$ (j = 2, ..., n). We repeat this step until we observe a distinct timing difference.

t_1 and t_2 are also used to recover the relative signs between the weights of the same row. Recall that the elements in $\mathbf{V}$s are absolute values. If $t_1 > t_2$, the two operands have opposite signs because the abnormal timing is triggered by the operation of $x - \Delta$. Otherwise, the two operands have the same signs.

So far, we have found the weight in the second column $\mathbf{V}_2$ that belongs to the same row as $\mathbf{V}_1[1]$. We repeat the process for other columns, and will get the row vector starting with $\mathbf{V}_1[1]$ and their relative signs. Finally, we repeat with the rest of the weights in Column 1, by setting $a = x/\mathbf{V}_1[i], i = 2, \cdots, n$. Eventually all the row vectors in $\mathbf{W_1}$ are recovered. This Algorithm involves three major loops; the complexity is $\mathcal{O}(mn^2)$.

3) Bias Vector: After the previous two attacks, we obtain a weight matrix $\mathbf{W}_1^{'}$. The order of the row vectors is possibly not the same as in $\mathbf{W}_1$. We can calculate the absolute values of the resulting vector of $\mathbf{W_1}^{'} \cdot \mathbf{l_0}$, which is all we need for the attack in this section to recover the actual signs and the bias vector.

We recover the actual signs of the weights by utilizing the characteristics of ReLU and observing the output layer. The ReLU function is defined as:

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

The output of ReLU will be zero as long as the input is negative. We can use this feature to determine if the value of a neuron before ReLU is negative. For example, we first define $\mathbf{l}^{'} = \mathbf{W_1}^{'} \cdot \mathbf{l_0}$. We control the input vector so that the absolute values of $\mathbf{l}^{'}$ are much larger than the absolute values of possible biases (e.g., 0.01[1]). Then, the signs of $\mathbf{l}^{'} + \mathbf{b_1}$ only depend on $\mathbf{l}^{'}$. We create another $\mathbf{l}^{''}$, which only differs from $\mathbf{l}^{'}$ in their first items (i'' vs. i') , which are not equal but have the same relative sign. Then we observe the outputs of the network corresponding to $\mathbf{l}^{'}$ and $\mathbf{l}^{''}$. If the two outputs are different, then the difference must come from the difference between the first items passing through the ReLU function. Hence, ReLU(i) $\neq$ ReLU($i^{'}$), and we know i and $i^{'}$ are positive (since same sign). Then we can recover the actual sign of the first row of the weight matrix. If the two outputs of the network are the same, then ReLU(i) = ReLU($i^{'}$), and we know i and $i^{'}$ are negative. We can repeat this for all the rows until we recover all the signs of the weight matrix.

So far, we have recovered $\mathbf{W}_1$ with all its weights (including the sign). We can recover the biases using a similar approach as the previous step. For an unknown operand x ($\mathbf{b_1}$) plus the other operand a ($\mathbf{W_1} \cdot \mathbf{l_0}$) that we control, we can incrementally scan a in the possible interval $[-b, b]$ with suitable stride size. By observing the result of ReLU($a+x$), we can recover x precisely when ReLU($a+x$) changes from positive to zero or zero to positive. I.e., we identify what value of a results in $a + x = 0$, and we can recover x.

[1]All models we trained and found online have weights and biases in the range $(-0.02, 0.02)$

C. Recovering the remaining layers

In this subsection, we discuss how to adapt the first-layer attack to the remaining layers. The key to extending the attack is that we can control the second layer input $\mathbf{l_1}$ with necessary granularity by only controlling the input layer $\mathbf{l_0}$. If we can achieve this, attacking the remaining layers is very similar to attacking the first layer. We therefore discuss the necessary control over $\mathbf{l_1}$ required for attacking the second layer, and evaluate the feasibility.

For recovering the absolute column values, we need to control one of the input elements and set the others to zero. To recover the rows, we need to control two of the input elements and set the others to zero. Zeros can be accomplished by setting the elements in $\mathbf{l} = \mathbf{W_1} \cdot \mathbf{l_0}$ to be large negative numbers to go through the ReLU function. We show how we can control $\mathbf{l_1}$ such that two elements can be set to a and b of our choice, and the others set to negative values.

The problem is formalized as the following claim: for a system $\mathbf{W_1} \cdot \mathbf{l_0} = \mathbf{l}$ of linear equations where $\mathbf{W_1}, \mathbf{l_0}, \mathbf{l}$ have dimensions $(n, m), (m, 1), (n, 1)$, respectively, there exists at least one $\mathbf{l_0}$ that can make $\mathbf{l}[i] = a, \mathbf{l}[j] = b, i, j \in [1, n], a, b \in \mathbb{R}, \mathbf{l}[k] < -\theta, k \neq i, j$, where θ is the largest bias possible. Here we assume $\mathbf{W_1}$ is a full-rank matrix, i.e., its the row vectors are linearly independent. We argue that considering the high precision of floating-point numbers, it is unlikely that the row vectors are linearly dependent. According to the theorem in [11], $\mathbf{l_0}$ exists if and only if $rank[\mathbf{W_1}] \geq rank[\mathbf{W_1}|\mathbf{l}]$. When $n \leq m+1$, the rank of $\mathbf{W_1}$ is n, and the rank of $\mathbf{W_1}|\mathbf{l}$ cannot be greater than n; When $n > m+1$ the rank of $\mathbf{W_1}$ is m. We can set $\mathbf{l}$ to be linearly independent from the column vectors of $\mathbf{W_0}$. Then, the rank of $\mathbf{W_1}|\mathbf{l}$ is at most m, and again we have found a value $\mathbf{l_0}$ as required.

IV. Experiments

In this section, we reverse-engineer an MLP model that classifies the MNIST dataset. We also evaluate both the accuracy of the recovered parameters and the accuracy of the recovered model. They both turn out to be very high, demonstrating the effectiveness of our novel floating-point timing-based side-channel attack.

A. Experimental Setup

The experimental platform is as discussed in subsection III-A: each timing difference in CPU cycles is measured for 100 times, and the most frequent ones are averaged.

The model we recovered is a four-layer MLP. The input layer flattens the MNIST dataset; hence it has a size of $28 \times 28 = 784$. The second and third layers both have a size of 50. The last layer is the output layer before the softmax function, which has a size of 10. All the activation functions are ReLU. The model is trained using stochastic gradient descent (SGD) with a learning rate of $1\text{e}{-2}$, a momentum of $5\text{e}{-1}$, and a batch size 64 for 5 epochs. The testing loss and accuracy are $1.342\text{e}{-1}$ and 96.04%, respectively. Our entire reverse-engineering attack takes less than one hour for the selected MLP model on our testing workstation.

B. Results

We first define the accuracy of a recovered parameter as: $\rho_p = 1 - \frac{|p-p'|}{p}$, where p' is the recovered parameter, and p is the original parameter. We evaluate the accuracies of all the recovered parameters in the first layer and take an average of them. We also evaluate the effect of the selected precision (ϵ in Algorithm 1) on the average accuracy of the first-layer parameters, as shown in Table III. When ϵ is smaller than $1\text{e}{-39}$ the accuracy is almost 1. We can set much smaller ϵ for our algorithm.

TABLE III
FIRST-LAYER PARAMETER ACCURACY WITH DIFFERENT ϵ

$-\log\epsilon$	37	38	39	40
ρ_p	0.838 ± 0.118	0.987 ± 0.011	0.998 ± 0.001	$0.999 \pm 1\text{e}{-4}$

We plug in the recovered model for testing with the MNIST dataset, and evaluate the model accuracy. Table IV shows the recovered model reaches the original testing accuracy when ϵ is smaller than $1\text{e}{-39}$.

TABLE IV
MODEL ACCURACY IN CLASSIFYING MNIST WITH DIFFERENT ϵ

$-\log\epsilon$	37	38	39	40
ρ_{model}	0.9193	0.9598	0.9604	0.9604

V. DISCUSSION

A. Platform Generality

IEEE 754 floating-point arithmetic exists in many modern processors, and potentially they may all have floating-point timing leakage. For example, we have observed abnormal timings of subnormal floating-point operations of ARM Cortext-A9 processors, though it is less pronounced than on Intel processors. We believe floating-point timing side-channels are subtle but pervasive and powerful, leading to complete DNN model recovery, as demonstrated in this work.

B. Convolutional Neural Networks

With our methodology, we can reverse-engineer other network layers like convolutional layers as well. Convolutional layers are usually implemented as matrix multiplication to take advantage of commodity math libraries. For example, a convolutional computation:

$$Cov\left(\begin{bmatrix} k_0 & k_1 \\ k_2 & k_3 \end{bmatrix}, \begin{bmatrix} l_{0,0} & l_{0,1} & l_{0,2} \\ l_{1,0} & l_{1,1} & l_{1,2} \\ l_{2,0} & l_{2,1} & l_{2,2} \end{bmatrix}\right)$$

is transformed to:

$$\begin{bmatrix} l_{0,0} & l_{0,1} & l_{1,0} & l_{1,1} \\ l_{0,1} & l_{0,2} & l_{1,1} & l_{1,2} \\ l_{1,0} & l_{1,1} & l_{2,0} & l_{2,1} \\ l_{1,1} & l_{1,2} & l_{2,1} & l_{2,2} \end{bmatrix} \cdot \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix}$$

Then we can adopt the same technique and timing model in III-B, recover k_0 by controlling the input $l_{0,0}$ and setting the other inputs to zero.

C. Future Work

We are also considering countermeasures. A straightforward one is to eliminate subnormal floating-point numbers. Subnormal computations can be disabled, at the expense of accuracy, which may or may not be sufficient given the application. However, normal floating-point operations leak information through timing side channels as well, even if the leakage is not as strong. We are currently investigating this as a possible attack surface.

VI. CONCLUSION

In this work, we present the first DNN reverse-engineering attack that utilizes the floating-point timing side-channel. This study proves that by combining the floating-point timing side-channel leakage and analyzing the output of model inference, one can recover all the parameters of an MLP model accurately. We also argue that this attack can potentially extend to many other CPU platforms and network structures.

REFERENCES

[1] "Stanford dawn deep learning benchmark (dawnbench) imagenet training," 2019. [Online]. Available: https://dawn.cs.stanford.edu/benchmark/ImageNet/train.html

[2] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *IEEE Symp. on Security & Privacy*, May 2017, pp. 3–18.

[3] A. Rozsa and T. E. Boult, "Improved adversarial robustness by reducing open space risk via tent activations," *arXiv preprint arXiv:1908.02435*, 2019.

[4] J. Breier, X. Hou, and et. al., "Practical fault attack on deep neural networks," in *Conf. on Computer & Communications Secruity*, Oct. 2018, pp. 2204–2206.

[5] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *Proc. Design Automation Conf.*, June 2018, pp. 4:1–4:6.

[6] M. Yan, C. Fletcher, and J. Torrellas, "Cache Telepathy: Leveraging shared resource attacks to learn DNN architectures," in *USENIX Security Symp.*, Aug. 2020.

[7] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *USENIX Security Symp.*, Aug. 2019, pp. 515–532.

[8] G. Dong, P. Wang, P. Chen, R. Gu, and H. Hu, "Floating-point multiplication timing attack on deep neural networs," in *IEEE Int. Conf. Smart Internet of Things*, 2019, pp. 155–161.

[9] A. Rane, C. Lin, and M. Tiwari, "Secure, precise, and fast floating-point operations on x86 processors," in *USENIX Security Symp.*, Aug. 2016, pp. 71–86.

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE Symp. on Security & Privacy*, May 2015, pp. 623–639.

[11] P. Suetin, A. I. Kostrikin, and Y. I. Manin, *Linear algebra and geometry*. CRC Press, 1989.