Stealing Neural Network Models through the Scan Chain: A New Threat for ML Hardware

Seetal Potluri
Department of ECE
North Carolina State University
Raleigh, U.S.
spotlur2@ncsu.edu

Aydin Aysu
Department of ECE
North Carolina State University
Raleigh, U.S.
aaysu@ncsu.edu

***Abstract*—Stealing trained machine learning (ML) models is a new and growing concern due to the model's development cost. Existing work on ML model extraction either applies a mathematical attack or exploits hardware vulnerabilities such as side-channel leakage. This paper shows a new style of attack, for the first time, on ML models running on embedded devices by abusing the scan-chain infrastructure. We illustrate that having course-grained scan-chain access to non-linear layer outputs is sufficient to steal ML models. To that end, we propose a novel *small-signal analysis* inspired attack that applies small perturbations into the input signals, identifies the quiescent operating points and, selectively activates certain neurons. We then couple this with a *Linear Constraint Satisfaction* based approach to efficiently extract model parameters such as weights and biases. We conduct our attack on neural network inference topologies defined in earlier works, and we automate our attack. The results show that our attack outperforms mathematical model extraction proposed in CRYPTO 2020, USENIX 2020, and ICML 2020 by an increase in accuracy of $2^{20.7}\times$, $2^{50.7}\times$, and $2^{33.9}\times$, respectively, and a reduction in queries by $2^{6.5}\times$, $2^{4.6}\times$, and $2^{14.2}\times$, respectively.**

I. Introduction

Stealing trained machine learning (ML) models is a significant challenge the industry faces given the exponential increase in model development costs [1]. Indeed, prior work has considered *model stealing/extraction* attacks in the context of ML-based cloud services [2]–[4], where the motivation is to exploit the commercial value of the ML model IP. Orthogonally, others have proposed using the extracted model to craft superior adversarial samples [5]–[7]. These works assume an ML-as-a-Service application with the prediction application programmer interface and publicly accessible query interfaces.

There is, however, a trend for edge intelligence where the trained ML algorithms execute on embedded devices, instead of on the cloud, to improve energy cost or response time. In this paper, we expose a new threat vector for model extraction on edge devices: *scan-chain attacks*. These attacks exploit the scan-chain infrastructure that allows in-field testing of deployed devices [8]–[10]. Although such attacks are well-known and well-recognized for cryptographic implementations [11]–[15], they have never been explored on ML hardware before.

We show that this extension is non-trivial—scan-chain access can be course-grained and neural networks have unique functions and floating-point arithmetic that do not occur in cryptography. Therefore, a successful and efficient attack requires innovative techniques to steal the weight and bias parameters of a neural network (NN) with a high precision and with a low number of queries.

A. Contributions

The contributions of this paper are as follows:

- We propose the first model stealing attack through scan-chain access. This attack exploits the *debug port* to observe the hardware internal states for model extraction.
- We show that the attack does not require accessing all internal registers. Instead, observing non-linear (i.e., activation layer or max-pooling layer) output is sufficient for the attack with two proposed novel techniques:
  1) Applying input perturbations and performing *small-signal analysis* to systematically expose the model parameters of each neuron.
  2) Formulating selective neuron activation as a linear constraint satisfaction (LCS) problem and using *coupled constraints* for $\{$*negative weight, negative bias*$\}$ cases while using *iterative constraint relaxation* for *hard cases* of the LCS problem.
- We show the attack's feasibility on models used in earlier works and defined for popular edge-device hardware IPs such as `Eyeriss` [16]. We automate the proposed attack.
- The results show that on average, our proposed attack is $2^{20.7}\times$, $2^{50.7}\times$, and $2^{33.9}\times$ more accurate and requires $2^{6.5}\times$, $2^{4.6}\times$, and $2^{14.2}\times$ fewer queries compared to the attacks proposed at CRYPTO 2020 [17], USENIX 2020 [18], and ICML 2020 [19] respectively.

Our work lays the theoretical foundations of scan-chain attacks on embedded ML hardware and quantifies its capability and superiority over cryptanalytic attacks.

B. Organization

The rest of the paper is organized as follows. The next section describes the threat model and Section III provides some background on NNs. Sections IV-(A) to (E) explain the proposed *small-signal analysis* inspired attack to extract the NN model parameters using the information obtained through scan-chains and Section IV-(F) describes automation in order to scale the attack to large NNs of arbitrary depths and sizes. Subsequently, Section V provides the results and comparison with prior works. Following this, Section VI provides a brief discussion of existing countermeasures, prior work, and how to switch between functional and debug modes and Section VII concludes the paper.

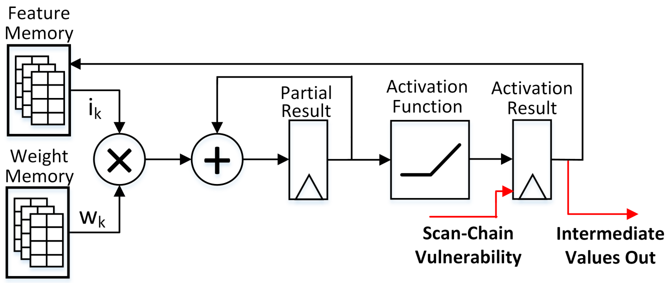


Fig. 1: Scan-chains enable accessing internal states and leaking intermediate results, which can be used to efficiently extract parameters.

II. Threat Model

We follow the standard threat model laid out in the seminal work of scan-chain attacks on cryptographic circuits [11] and detailed in subsequent works [12]–[15]. In this context, companies that provide independent semiconductor assembling and test manufacturing services have a multi-billion dollar market in 2020 [20]. Their system-level-test (SLT) [21] capability translates to the adversary's ability to test the chip that contains the NN models, both in the functional and scan modes of operation. We assume that the adversary is such a third-party testing service provider who has access to scan chains. Additionally, following earlier work [11], [22], [23], we assume that the adversary can find the positions of all scan elements in the scan-chain through semiconductor reverse engineering [24].

The adversary targets the inference and its goal is to extract trained model parameters such as weights and biases of a deep neural network. We consider an adversary having physical access to an edge device that performs the inference. The adversary executes the classification for a certain number of functional cycles, switches to debug mode, and uses the hardware debug port of the edge device to leak internal states as shown in Figure 1. The adversary then performs a statistical analysis on these internal states to steal the NN model parameters.

There are dedicated registers for storing the configuration bits in neural network hardware accelerators [16]. These configuration registers are tamper-proof, thus making it infeasible for the adversary to ascertain the contents of the partial results. By contrast, the contents of the `ReLU` activation-unit output register can be ascertained because, after a fixed number of clock cycles, it will contain the output of a certain neuron in a certain layer, before sending it to the DRAM [16]. Therefore, our attack only assumes access to the activation results shown in Figure 1.

Recent work categorizes model extraction attacks into high fidelity vs. high accuracy [18]: our adversary performs a *high fidelity* attack that aims for general agreement between the extracted and victim models on any input. Following earlier works on model extraction [4], [18], [25], we assume the hyper-parameters of the model like the types/number of neural network layers are either public or obtained by another attack [6], [26].

III. Background

A neural network (NN) can be described as a series of functional transformations, where the architecture and parameters completely describe the NN model. A training set is used to train/tune the parameters of an adaptive NN model. Once the training is complete, the network is used to perform classification or regression. This work focuses on NN models for classification. A fully connected neural network (FCNN) is a feed-forward neural network, consisting of fully connected layers which connect every neuron in one layer to every neuron in another layer. Figure 2 shows two hidden layers of an FCNN. The outputs of neurons in $(i+1)^{th}$ $(0 \leq i < d)$ fully connected layer can be described as:

TABLE I. Glossary of Symbols

Symbol	Definition
TMS	Test Mode Signal
SE	Scan-Enable Signal
f	PE filter row-size
p	Accelerator array size
W_i	Weight vector size for neuron in layer-(i)
$T(i)$	Computation time of a neuron in layer-(i)
$T_c(i)$	Cumulative computation time of a neuron in layer-(i)
d	Depth of the neural network
N_I	Input dimension
N_i	Number of neurons in layer-(i)
$h(.)$	Non-linear activation function
$\mathbf{x}^i$	Input to layer-(i)
$\mathbf{x}^{(i+1)}$	Output of layer-(i)
$\mathbf{W}^i$	Weight matrix of layer-(i)
$\mathbf{W}^i$	Weight matrix of layer-(i)
$\mathbf{W}^i[m]$	Weight vector of $(m+1)^{th}$ neuron in layer-(i)
$w^i[j][k]$	$(k+1)^{th}$ weight of neuron-(j) in layer-(k)
$\mathbf{b}^i$	Bias vector of layer-(i)
$\mathbf{b}^i[k]$	Bias of k^{th} neuron in layer-(i)
$\mathbf{I}$	Input to the neural network
$O^i[m]$	Output of $(m+1)^{th}$ neuron in layer-(i)
δ_k	Perturbation to k^{th} element in the input $\mathbf{I}$
$\triangle$	Perturbation amplitude

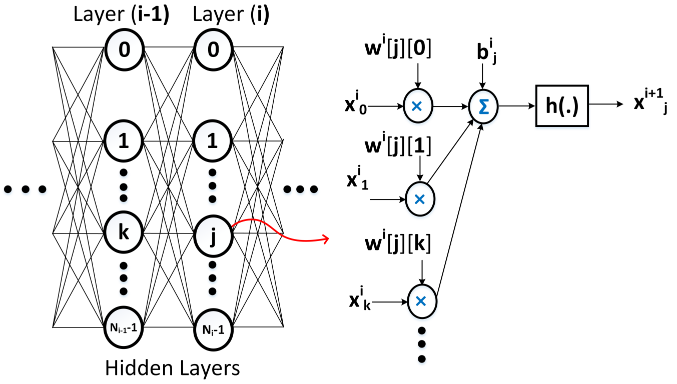


Fig. 2: The connections between two hidden layers of an FCNN

$$\mathbf{x^{i+1}} = h\left(\mathbf{W^i}.\mathbf{x^i} + \mathbf{b^i}\right) \tag{1}$$

where $h(.)$ is the non-linear activation function, and d is the number of hidden layers (refer Table I). Here, $\mathbf{x^i} = [x^i_0 \; x^i_1 \ldots x^i_k \ldots x^i_{(D-1)}]^T$ (column vector) is the input vector to layer-(i), where each variable x^i_k corresponds to the $(k+1)^{th}$ input feature to the layer and

$$D = \begin{cases} \mathrm{N}_I & \text{i=0} \\ \mathrm{N}_{(i-1)} & \mathrm{i} \neq 0 \end{cases}$$

where N_I is the input dimension and $N_{(i-1)}$ is the number of neurons in layer-$(i-1)$ (prior layer). Similarly, $\mathbf{x^{i+1}} = [x^{i+1}_0 \; x^{i+1}_1 \ldots x^{i+1}_j \ldots x^{i+1}_{(N_i-1)}]^T$ (column vector) is the output vector of layer-(i), where N_i is the number of neurons in layer-(i) (current layer) and each variable x^{i+1}_j corresponds to the output of neuron-(j) in layer-(i). The secret model parameters are the weight matrix coefficients, where $\mathbf{W^i} = \{w^i[j][k]\}$ is the weight matrix of layer-(i), whose $(j+1)^{th}$ row corresponds to the weight vector of neuron-(j) in layer-(i) and bias coefficients, where $\mathbf{b^i} = [b^i_0\ b^i_1 \dots b^i_j \dots b^i_{(N_i-1)}]^T$ (column vector) is the bias vector of layer-(i), where b^i_j is the bias value of neuron-(j) in layer-(i), as shown in Figure 2. These weights and biases are obtained during training and is then constant during inference. The weights, biases, and activation values are represented as double-precision floating-point (DPFP) numbers in this work.

The rectified linear unit (ReLU) [27] is the non-linear activation function $h(.)$ we used. This is due to ReLU's widespread success and to be consistent with the earlier works on cryptanalytic model extraction [17], [18]. The ReLU activation function is given below where x is the output of a neuron in a fully connected layer and $h(x)$ is the activation result:

$$h(x) = max(0,x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

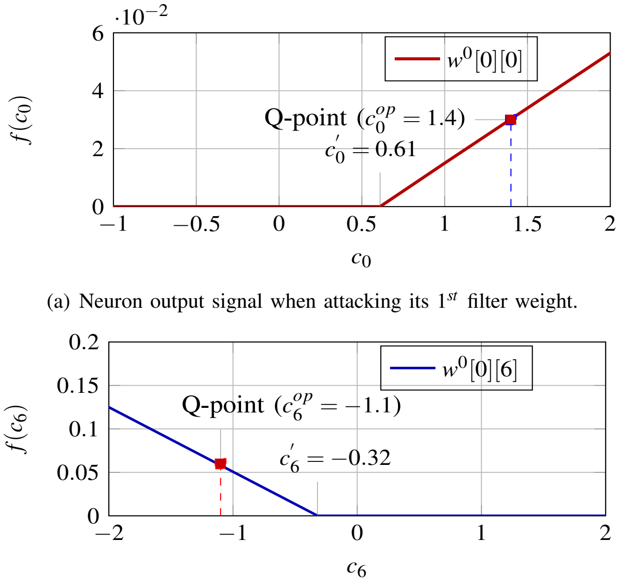
The output layer uses *softmax* operation for generating the confidence scores and subsequently selecting the output class with the maximum confidence score. During inference, the output layer performs an order-preserving transformation on the output signals of the penultimate layer [17], [18] and uses a $max(.)$ operation over these transformations, and selects the corresponding index as the output of the classifier.

IV. The Proposed Attack

To steal NN model parameters, we apply an attack inspired by *small-signal analysis* (SSA) used to extract analog system model parameters like gain and transconductance [28]. We use SSA to identify the linear region of operations for neurons where the ReLU returns a non-zero value. We then use this to apply small perturbations into neural network input signals to generate a system of linear constraints (SLC) with known inputs and unknown parameters and solve the system using a *linear constraint solver*. We address the challenges of exposing multiple weights and the parameters in subsequent layers through an iterative procedure. The iterative procedure (a) uses the extracted parameters of the previous hidden layers to generate the SLC for the current hidden layer, (b) solve the SLC to identify the neuron quiescent operating point (*Q-point*), (c) apply small perturbations into neural network input signals around the *Q-point*, let them propagate until the current hidden layer, and (d) use the black-box responses of the current hidden layer to extract the model parameters through SSA. We finally automate this iterative procedure to extend the proposed attack to FCNNs of arbitrary topologies and depths.

A. Identifying neuron quiescent operating point (Q-point)

The goal of *Q-point* identification is to search for an input that activates desired neurons and deactivates the remaining neurons in a stable way. In this context, stability refers to the activated neurons remaining active and the inactive neurons remaining inactive even in the presence of small-signal perturbations at the input. ReLU makes inactive neuron to always produce a "0" output, which does not provide useful information about the NN model. By contrast, an activated neuron's output is a linear function of its model parameters, which leaks information and can be extracted. Such carefully selected inputs, popularly known as quiescent operating points (*Q-points*) in the analog signal processing literature, are coupled with *SSA* to extract circuit model parameters. Our model extraction is inspired by this strategy. This section describes *Q-point* identification/search.



(a) Neuron output signal when attacking its 1^{st} filter weight.

(b) Neuron output signal when attacking its 7^{th} filter weight.

Fig. 3: Varying neuron signal behaviors, depending on the **weight-under-attack**. Please note the Y-axis range.

The weighted summation operation of a neuron can be considered as a filtering operation, and the array of weights can be considered as a neural filter [16]. Suppose we are interested in the output signal of the first neuron in the first hidden layer of an FCNN with the filter defined as:

$$\mathbf{W}^0[0] = \left[\begin{array}{cccc} w^0[0][0] & w^0[0][1] & \dots & w^0[0][N_I-1] \end{array}\right]$$

If we are interested to attack $w^0[0][0]$ (the superscript "0" denotes the first hidden layer, the first index "0" indicates the first neuron in the layer, and the second index "0" indicates the first weight in the filter), we can make all the entries in the input vector $\mathbf{I}$ zero except the first entry containing a user-defined variable c_0, represented as:

$$\mathbf{I} = \left[\begin{array}{cccc} c_0 & 0 & \dots & 0 \end{array}\right]$$

We refer to the classifier as an *oracle*, to which we make queries and record the corresponding internal states. If we make a query to the oracle by applying this input and observe output of first neuron in the first layer ($O^0[0]$) through the debug port, we notice that the signal is purely a function of c_0:

$$f(c_0) = O^0[0] = ReLU(c_0.w^0[0][0] + b^0[0]) \qquad (2)$$

Figure 3(a) shows the response of neuron-(0) in layer-(0) for varying input (c_0). The example reflects the case where an FCNN trained on MNIST and mapped to an accelerator (e.g., `Eyeriss` [16]), where our attack makes oracle queries with different values of c_0 and observes the debug port output. We make the following observations from this signal characteristic:

- The neuron output signal is "0" for $c_0 < 0.61$ and behaves as a linear function of the weight $w^0[0][0]$ with any further increase in c_0. This threshold is denoted as $c_0^{'}$.
- Positive perturbations to c_0 around $c_0^{'}$ activate the neuron, while negative perturbations do not. Thus, c_0 is sensitive

around $c_0^{'}$ and the operating point of c_0 should be set to a value reasonably higher than $c_0^{'}$ so that small perturbations either way (positive or negative) do not disturb the linear behavior of the neuron.

If we denote the operating point of c_0 as c_0^{op}, then how greater it has to be as compared to $c_0^{'}$ depends on the magnitude of the small-signal. If we denote small-signal on c_0 as δ_0 and the amplitude of δ_0 as $\triangle$, then we can safely define the operating point as $c_0^{op} = c_0^{'} + \beta.\triangle$, where $\beta > 1$, is a tuning parameter used to adjust the *Q-point* to the stable region of operation (i.e., both positive and negative perturbations with amplitude $\triangle$ on c_0 around the *Q-point* are acceptable). By this definition of operating point, we know for sure that $f(c_0) = f(c_0^{op} \pm \delta_0) > 0$, hence forcing the neuron to operate in the linear region of operation. Although we have so far explained the procedure to extract $w^0[0][0]$, the methodology is clearly generic. Figures 3(a) and 3(b) show the stable neuron operating points in the linear region, when attacking weights $w^0[0][0]$ (1^{st} weight in the filter of neuron-(0) in layer-(0)) and $w^0[0][6]$ (7^{th} weight in the filter of neuron-(0) in layer-(0)) respectively.

B. Small-Signal Analysis (SSA)

The connection between two neurons has two unknown variables: *weight vector* and *bias*, where *bias* is fixed for all the connections to a neuron in the current hidden layer. Therefore, obtaining two linear equations is sufficient to solve and extract an *unknown weight* for a *given neuron*. After computing a stable neuron operating point (*Q-point*) in the linear region of operation, we achieve this with an *SSA* that consist of primarily two phases:

1) Applying input **I** to the neuron (since it is the first hidden layer, **I** is directly applied to the neuron input; for the subsequent hidden layers, a transformed version of **I** gets applied) and computing its output.
2) Applying a small perturbation $\delta < \triangle$ to **I** to produce $\mathbf{I}^p$, applying $\mathbf{I}^p$ to the neuron and computing its output.

The results computed in these two stages will be useful for the extraction of model weights and biases. In our example, we obtain $f(c_0^{op})$ and $f(c_0^{op} + \delta_0)$ in these two phases. The difference between these two signals gives:

$$\begin{aligned} f(c_0^{op} + \delta_0) - f(c_0^{op}) &= ReLU((c_0^{op} + \delta_0).w^0[0][0] + b^0[0]) \\ &\quad - ReLU(c_0^{op}.w^0[0][0] + b^0[0]) \\ &= ((c_0^{op} + \delta_0).w^0[0][0] + b^0[0]) \\ &\quad - (c_0^{op}.w^0[0][0] + b^0[0]) = \delta_0.w^0[0][0] \quad (3) \end{aligned}$$

Hence $f(c_0^{op} + \delta_0) - f(c_0^{op}) = \delta_0.w^0[0][0]$ (this has become plausible because of the linear operating point). Thus, the filter weight can extracted as:

$$w^0[0][0] = \frac{f(c_0^{op} + \delta_0) - f(c_0^{op})}{\delta_0} \quad (4)$$

If we substitute this value for $w^0[0][0]$ in Equation 2, we obtain the bias using:

$$b^0[0] = f(c_0^{op}) - c_0^{op}.w^0[0][0] = (1 + \frac{c_0^{op}}{\delta_0}).f(c_0^{op}) - (\frac{c_0^{op}}{\delta_0}).f(c_0^{op} + \delta_0) \quad (5)$$

The above calculations return $b^0[0] = -2.36e^{-2}$ and $w^0[0][0] = 3.85e^{-2}$, which are indeed the correct values for the target FCNN. Thus, we have successfully and accurately obtained both the filter weight and neuron bias. Note that this is possible given scan-chain access to the output of the non-linear layer of activation results. In our example, we have chosen the non-zero entry in location-(0) of the input signal **I**, hence we are able to extract the 1^{st} filter weight in neuron-(0) $w^0[0][0]$ successfully. In fact, we can choose any location-(k) in neuron-(j) and successfully extract corresponding filter weight $w^0[j][k]$, $\forall\ j \in [0, N_0)$, $\forall\ k \in [0, N_I)$. We will show later in Section IV-D, that this can indeed be generalized to other layers as well.

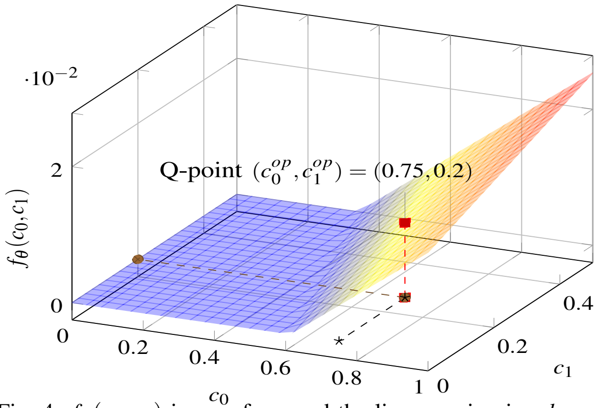


Fig. 4: $f_\theta(c_0, c_1)$ is a surface, and the linear region is a *hyperplane* in the 3-dimensional space.

C. Exposing multiple filter weights using a common Q-point

So far, for simplicity, we have seen the analysis corresponding to exposing one filter weight at a time. Although this method works well for the {*positive weight, positive bias*} (case-1), {*positive weight, negative bias*} (case-2) and {*negative weight, positive bias*} (case-3) cases, this method does *not* extend to the {*negative weight, negative bias*} (case-4) case in the subsequent hidden layers[1]. This is because all subsequent layers receive only non-negative inputs due to the ReLU activation. To address this limitation, it is important to find a *common Q-point* for multiple weights inside a neuron, and then expose the individual weights. Consider the simple example of applying the following input with only two non-zero entries at locations (0) and (1):

$$\mathbf{I} = \begin{bmatrix} c_0 & c_1 & 0 & \dots & 0 \end{bmatrix}$$

In this scenario, the output signal (of neuron-(0) in layer-(0)) is a function of both c_0 and c_1:

$$f(c_0, c_1) = O^0[0] = ReLU(c_0.w^0[0][0] + c_1.w^0[0][1] + b^0[0]) \quad (6)$$

Since $f(c_0, c_1)$ is a rectified linear function of the 2 user-defined variables c_0 and c_1, the neuron output function is a surface and the linear region is a *hyperplane*, both in 3-dimensional space as shown on Figure 4. Going a step forward, if we allow as many non-zero entries as the number of filter weights at corresponding locations, then $\mathbf{I} = \mathbf{C}$ where

$$\mathbf{C} = \begin{bmatrix} c_0 & c_1 & \dots & c_{(N_I - 1)} \end{bmatrix}$$

then the corresponding neuron output signal is given by:

$$f(\mathbf{C}) = O^0[0] = ReLU(\mathbf{C} \odot \mathbf{W^0[0]} + b^0[0]) \quad (7)$$

[1] It was possible for the case shown in Figure 3(b), because negative inputs are possible for the first hidden layer.

where $\odot$ is the element-by-element (scalar) multiplication between the vectors. Here also, $f(\mathbf{C})$ is a rectified linear function of the elements in $\mathbf{C}$. If n be the dimension of the vector space for $\mathbf{C}$, then $f(\mathbf{C})$ is a surface and the linear region is a *hyperplane*, both in $(n+1)$-dimensional space.

Let $\mathbf{C}^{op}$ be the linear operating point (*Q-point*), and let $\mathbf{C}^p(k)$ be the perturbed vector with perturbation only in location-(k). Suppose we are interested in attacking weight $w^0[0][0]$, we can then define $\mathbf{C}^p(0)$ as:

$$\mathbf{C}^p(0) = \left[\begin{array}{cccc} c_0 + \delta_0 & c_1 & \dots & c_{(N_I-1)} \end{array} \right]$$

In that case, the weight $w^0[0][0]$ can be extracted as follows:

$$\begin{aligned} f(\mathbf{C}^p(0)) - f(\mathbf{C}^{op}) &= ReLU(\mathbf{C}^p(0) \odot \mathbf{W^0}[\mathbf{0}] + b^0[0]) \\ &\quad - ReLU(\mathbf{C}^{op} \odot \mathbf{W^0}[\mathbf{0}] + b^0[0]) \\ &= (\mathbf{C}^p(0) \odot \mathbf{W^0}[\mathbf{0}] + b^0[0]) \\ &\quad - (\mathbf{C}^{op} \odot \mathbf{W^0}[\mathbf{0}] + b^0[0]) = w^0[0][0].\delta_0 \end{aligned} \tag{8}$$

In Equation 8, the `ReLU` function disappears because the neuron is operating around *Q-point*, and is thus in linear region of operation. This means the filter weight can be calculated as:

$$w^0[0][0] = \frac{f(\mathbf{C}^p(0)) - f(\mathbf{C}^{op})}{\delta_0} \tag{9}$$

All weights within the filter $\mathbf{W}^0[0]$ can be extracted similarly:

$$w^0[0][k] = \frac{f(\mathbf{C}^p(k)) - f(\mathbf{C}^{op})}{\delta_k}, 0 \le k < D \tag{10}$$

Subsequently, the neuron bias for the neuron-(0) in layer-(0) can be extracted using:

$$b^0[0] = f(\mathbf{C}^{op}) - \mathbf{C}^{op} \odot \mathbf{W}^0[0] \tag{11}$$

This can further be generalized to all neurons in layer-(0).

D. Exposing model parameters beyond the first hidden layer

In the first layer, the entries of the input and the filter have a one-to-one mapping hence exposing exactly one weight through SSA was possible. By contrast, the small-signal of any given index in the input $\mathbf{I}$ gets broadcasted to all inputs to the second layer. For example, if we apply a small-signal δ_0 to c_0, we know that $O^0[0]$ will be a function of δ_0. Similarly, $\{\mathbf{O^0[j]}\}$, $0 \le j < N_0$ are all functions of δ_0, thereby each weight of a given neuron in the second layer gets multiplied by a function of δ_0. This not only restricts direct exposing of weights like the first layer but also results in an *under-determined system* of linear equations, hence difficult to solve.

1) Under-determined system of equations: If we consider again the single perturbation scenario:

$$\mathbf{C}^p(0) = \left[\begin{array}{cccc} c_0 + \delta_0 & c_1 & \dots & c_{(N_I-1)} \end{array} \right]$$

The SSA analysis on $(j+1)^{th}$ input to any second layer neuron is given as $\bigtriangledown \mathbf{C}^p_{1,j} = \delta_0.w^0[j][0]$. The SSA output of neuron-(0) in layer-(1), as a function of its input features and filter weights, can be computed as:

$$\begin{aligned} \bigtriangledown O^1[0] &= \sum_{j=0}^{N_0-1} (\bigtriangledown \mathbf{C}^p_{1,j}) \times w^1[0][j]) = \sum_{j=0}^{N_0-1} (\delta_0.w^0[j][0]) \times w^1[0][j]) \\ &= \delta_0 \cdot \sum_{j=0}^{N_0-1} (w^0[j][0]) \times w^1[0][j]) \end{aligned} \tag{12}$$

Thus, independent of δ_0, it results in exactly one equation, which is not sufficient to solve for N_0 unknowns. Going further, if we consider the multiple perturbation scenario:

$$\mathbf{C}^p(0,1,\dots,(N_I-1)) = \left[\begin{array}{ccc} c_0 + \delta_0 & c_1 + \delta_1 \dots & c_{(N_I-1)} + \delta_{(N_I-1)} \end{array} \right]$$

it is possible to generate N_I linearly-independent equations, but this system is also under-determined if $N_I < N_0$.

2) Solution: This challenge can be solved through *localization*, where we place exactly one neuron in layer-(0) in the linear region of operation. For example, if we are interested in extracting $w^1[0][j]$ (the $(j+1)^{th}$ weight in $\mathbf{W}^1[0]$, i.e., the filter corresponding to neuron-(0) in layer-(1)), we need to search for a *Q-point* that **does** activate only the $(j+1)^{th}$ neuron in layer-(0), and **does not** activate the remaining neurons in layer-(0). In that case, Equation 12 simplifies to:

$$\bigtriangledown O^1[0] = \delta_0.(w^0[j][0] \times w^1[0][j]) \tag{13}$$

and hence the unknown weight in the second layer neuron can be extracted using the known values as:

$$w^1[0][j] = \frac{\bigtriangledown O^1[0]}{(\delta_0.w^0[j][0])} \tag{14}$$

In general, any weight $w^1[m][j]$ (the weight at the $(j+1)^{th}$ position of $\mathbf{W}^1[m]$, i.e., the filter corresponding to the $(m+1)^{th}$ neuron in the second layer) can be extracted/attacked in three steps.

1) Finding *Q-point* that activates only the $(j+1)^{th}$ neuron in the first layer, and does not activate remaining neurons in the first layer.
2) Applying perturbation δ_k at any arbitrary location-(k) in the input $\mathbf{I}$ to the neural network.
3) And subsequently observing $O^1[m]$, i.e., the activated output of $(m+1)^{th}$ neuron in the second layer.

and finally extracting the model weight using:

$$w^1[m][j] = \frac{\bigtriangledown O^1[m]}{(\delta_k.w^0[j][k])} \tag{15}$$

Although this subsection discusses the second layer, this *localization* method to extract model weights is generic and can be extended to any layer of the FCNN, *iteratively,* by searching for a *Q-point* that activates only one neuron in the prior layer. Once the weights are extracted, the biases can be successfully extracted similar to the technique used in Equation 11.

E. Linear Constraint Satisfaction

The activation of exactly one neuron in the first layer corresponding to the weight-under-attack (WUA) in the second layer neuron is basically a *constraint satisfaction* problem. The neuron function is itself linear, hence the constraint corresponding to the WUA index is a linear constraint with > *inequality* and constraints for the remaining weights are also linear with ≤ *inequality*. Hence, all the constraints are linear. Thus, we can formulate the problem of selective neuron's activation as a linear constraint satisfaction (LCS) problem[2]. In order to extract model parameters with high fidelity corresponding to cases-(1),(2),(3),

[2] Without SLC formulation, the results will not converge. A random search on *IBM BladeCenter® High-Performance Cluster* visited > 10 *billion* solutions occupying 1*TB* of disk-space without convergence after 1 week of execution.

we will need to generate SLC that activates only one neuron in the prior layer (layer-(0)), and can be described as:

$$\begin{aligned} \mathbf{W}^0[j].\mathbf{I}^\intercal &> -1*b_j+\eta, && j=j_{WUA} \\ \mathbf{W}^0[j].\mathbf{I}^\intercal &\le -1*b_j+\alpha, && j\in[0,N_0-1)-\{j_{WUA}\} \end{aligned} \tag{16}$$

However, in order to extract model parameters corresponding to case-(4), since isolated activation leads to the oracle response being "0", we will need to generate SLC that activates multiple neurons in the prior layer and reformulate on-the-fly as:

$$\begin{aligned} \mathbf{W}^0[j].\mathbf{I}^\intercal &> -1*b_j+\tau, && j=j_{WUA},\tau\in[\nu,\kappa] \\ \mathbf{W}^0[j].\mathbf{I}^\intercal &> -1*b_j+\eta, && j=j_{fp},\eta\gg\kappa \\ \mathbf{W}^0[j].\mathbf{I}^\intercal &\le -1*b_j+\alpha, && j\in[0,N_0-1)-\{j_{fp},j_{WUA}\} \end{aligned} \tag{17}$$

Here fp corresponds to the first positive weight in the layer-(1) neuron and $\nu<\kappa<<\eta$ to ensure the effect of negative weight will not dominate that of the positive weight, and hence ensure ReLU activation. This analysis is generic and extends to any layer-(i). Thus, for each neuron in layer-(i), (a) we generate an SLC for selective neuron activation in the layer-($i-1$), (b) solve the SLC, (c) extract the solution and make oracle queries to capture outputs of layer-(i) through scan-chains and finally (d) apply SSA to extract the model parameters of layer-(i). These steps need automation to enable *model-extraction* of large FCNNs of arbitrary depths and sizes.

F. Automation

Algorithm 1[3] shows the proposed automation to extract $\hat{\theta}_{(i)}$ (model-subset for layer-(i)), given $\hat{\theta}_{(i-1)}$ (model-subset for already extracted layer-($i-1$)). Apart from $\hat{\theta}_{(i-1)}$, the algorithm also accepts other user-tunable parameters δ (the small-signal), η (target for cases-(1),(2),(3)), $[\nu,\kappa]$ (target range for case-(4)), α (to prevent numerical errors due to the neurons being at the boundary between deactivation and activation). During our experiments, we have also encountered *hard cases* (case-(5)) when the solver could not successfully solve some of the SLC instances and returns "Infeasible".

We note that the algorithm has different procedures for different cases. For cases-(1), (2), (3) (discussed in Section IV-C), isolated activation is used where we use an exclusive *Q-point* for higher fidelity. For case-(4), we are forced to use a common *Q-point*. During our experiments, we have also encountered *hard cases* (case-(5)) when the solver failed to solve some of the SLC instances and returns "Infeasible". To address case-(5), we have devised a strategy to relax constraints in Equations 16 and 17 for values of i corresponding to weights already successfully extracted. This is acceptable, because if those model parameters appear in Equation 12, we can substitute the extracted parameters and solve for the unknown parameters.

V. Experimental Setup and Results

To evaluate and compare our results with earlier work using mathematical model extraction, we follow the approach of Carlini *et al.* [17]. We build the same neural network configurations with the same training process, apply our automated attack, and report the number of queries used and the maximum error rate ($max|\theta-\hat{\theta}|$) of the extracted parameters. This will effectively extend their results table and compare our work with the three earlier attacks proposed at CRYPTO 2020 [17], USENIX 2020 [18], and ICML 2020 [19].

Algorithm 1: Algorithm to attack target layer in FCNNs

```
Input: oracle, $\hat{\theta}_{(i-1)}$, $\delta$, $\eta$, $\kappa,\nu$ ($\eta\gg\kappa$)
$m := 0$;
while $m < N_i$ do
    $j := 0$;
    while $j < N_{(i-1)}$ do
        Generate constraints that activate only neuron-($j$) in
          layer-($i-1$) with $\eta$ as target activation;
        Run constraint-solver for Q-point search;
        // Cases-(1),(2),(3)
        Apply $SSA(oracle,Q,\delta)$ to extract $w^i[m][j]$ weight;
        Mark index of first positive weight of neuron-($m$) in
          layer-($i$) as $fp$;
        // Coupled Constraints for case-(4)
        if extracted weight = 0.0 then
            Generate constraints that activate neuron-($j$) with
              target activation $\tau$ ($\{\nu\le\tau\le\kappa\}$) and
              neuron-($fp$) with target activation $\eta$ (and none
              of the others) in layer-($i-1$);
            Run constraint-solver for Q-point search;
            Apply $SSA(oracle,Q,\delta)$ to extract the negative
              weight.
        end
        $j := j+1$;
    end
    // Iterative constraint relaxation for
      case-(5)
    $j := 0$;
    while $j < N_{(i-1)}$ do
        if Constrained solution "Infeasible" then
            Relax constraints for already solved weights;
            Run constraint-solver for Q-point search;
            Apply Incremental extraction() along with
              $SSA(oracle,Q,\delta)$ to extract the weight.
        end
        $j := j+1$;
    end
    Apply SSA to extract bias of neuron-($m$) in layer-($i$);
    $m := m+1$;
end
Result: $\hat{\theta}_{(i)}$
```

Our C implementation exploits the register-level information, by simulating the neural network accelerator functions, similar to the Eyeriss Chip Simulator [29]. We have used TensorFlow Graphics Processing Unit (GPU) as the backend, with a Keras front end to train and evaluate the accuracy of the FCNN models. We train and evaluate FCNN models on a computer with 64 GB of Random Access Memory (RAM), an NVIDIA 2080 Ti graphics card, and an Intel i7 9700K Processor.

We use a categorical cross-entropy as our loss function with Adagrad as the optimizer with a learning rate of 0.1. Using Adagrad allows the learning rate to be adjusted dynamically as the model is training. Thus, even with a large learning rate, the optimizer will update it to the appropriate value. The average time to train the models is around 10 minutes to achieve over 99% accuracy. We have used MNIST digit-recognition database [30] to train the 784-32-1 and 784-128-1 architectures (first two entries of Table II). The remaining architectures are benchmarks borrowed from [17].

The input Q-points are calculated using the ILOG constraint solver. Our C implementation invokes the solver and applies

[3] Simplified notation compared to Section IV-E.

TABLE II. Efficacy of our extraction attack, which is orders of magnitude more precise and query-efficient than prior work. Models denoted $a-b-c$ are fully connected neural networks with input dimension a, one hidden layer with b neurons, and c outputs. Entries denoted with a † were unable to recover the network after ten attempts. Here, θ and $\hat{\theta}$ denote the original and extracted FCNN models respectively.

Architecture	Parameters	Approach	Queries	$max\|\theta - \hat{\theta}\|$
784-32-1	$25,120$	[18]	$2^{18.2}$	$2^{-1.7}$
		[17]	$2^{19.2}$	$2^{-30.2}$
		Ours	$\mathbf{2^{13.9}}$	$\mathbf{2^{-53.3}}$
784-128-1	$100,480$	[18]	$2^{20.2}$	$2^{-1.8}$
		[17]	$2^{21.5}$	$2^{-29.4}$
		Ours	$\mathbf{2^{15.4}}$	$\mathbf{2^{-49}}$
10-10-10-1	210	[19]	2^{22}	2^{-12}
		[17]	2^{16}	2^{-36}
		Ours	$\mathbf{2^{8.8}}$	$\mathbf{2^{-45.9}}$
10-20-20-1	420	[19]	2^{25}	∞†
		[17]	$2^{17.1}$	2^{-37}
		Ours	$\mathbf{2^{10.2}}$	$\mathbf{2^{-45.4}}$
40-20-10-10-1	$1,110$	[17]	$2^{17.8}$	$2^{-27.1}$
		Ours	$\mathbf{2^{11.1}}$	$\mathbf{2^{-43.5}}$
80-40-20-1	$4,020$	[17]	$2^{18.5}$	$2^{-39.7}$
		Ours	$\mathbf{2^{13}}$	$\mathbf{2^{-44.2}}$

the RLC uncompressed versions of Q-points calculated by the solver iteration of Algorithm-1 through system calls similar to "IO2.py" of EyerissF [29], and reads back the calculated Q-points using *DPFP* representations. We have performed neuron-by-neuron and layer-by-layer computation through iterative calls to *oracle_nn_lx*(...) functions in a cycle-accurate fashion, similar to "EyerissF.py" which iteratively reuses the PE class defined in "PE.py".

Our implementation captures the activation results (shown in Figure-1) similar to "Activation.py" of EyerissF. The configuration parameters including the number of layers, number of neurons per layer, number of weights per neuron, and other parameters like $\alpha, \eta, \delta, \kappa, \nu$ are defined as macros, similar to "conf.py" of EyerissF.

Table II presents the results of our attack and its comparison with the mathematical model extraction. The "parameters" column in Table II indicates the number of parameters in the NN model. The results clearly demonstrate the superiority of our attack across all the considered architectures. On average, our proposed attack is $2^{20.7}\times$, $2^{50.7}\times$ and $2^{33.9}\times$ more accurate and requires $2^{6.5}\times$, $2^{4.6}\times$ and $2^{14.2}$ less queries compared to [17], [18], and [19] respectively.

The results depend on the user-tunable parameters η, κ, ν, while the dependence on δ is observed to be small. We used the `IBM® ILOG CPLEX` *constraint-solver* for solving the SLCs. Algorithm 1 was implemented in `C` and the *constraint-solver* was invoked through system-calls. The results shown in Table II correspond to $\delta = 1.0$, $\alpha = -1.0$, $\eta = 10.0$, $\kappa = 2.0$ and $\nu = 1.0$—these values are constant across all the FCNN architectures. The accuracy can be further improved by a suitable choice of these parameters, which we leave for future investigation.

Since `ReLU` also uses *max*(.) operation, we assume `ReLU` activation-unit [16] can be re-used to compute the output layer, and hence the output of the last layer also leaks information. We have contacted authors of prior papers [17], [18] and confirmed that our assumption on the output layer is consistent with theirs.

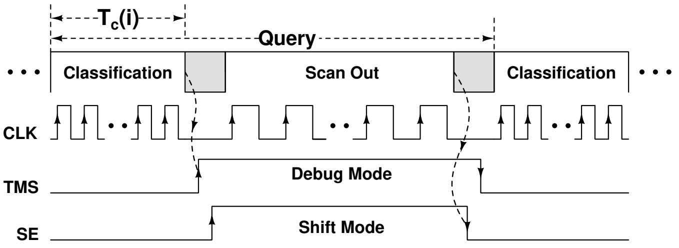


Fig. 5: Timing diagram for switch between functional and debug modes

Although it is an orthogonal threat, an interesting comparison of our attack can be made with the side-channel analysis, which seems to be more powerful as it can directly attack individual floating-point multiplications. It is reported that a few thousand queries are sufficient to extract the mantissa but direct comparison is hard because the attack was conducted on single-precision floating-point variables and the maximum error rate and queries are not given for the complete network [31].

VI. Discussion

This section provides a brief discussion on existing countermeasures, prior work, and how to switch between functional and debug modes to extract layer-wise information.

A. Existing Countermeasures on Cryptographic Accelerators

It is possible that existing countermeasures on cryptographic accelerators such as *Secure Scan* [32], *Mode-reset* [33], *Scan-cell/Output Swapping* [34] etc. can prevent our proposed attacks but such defenses have to be adapted/implemented for the new ML accelerators and the designers have to accept the related overheads. But the first step is to show that such attacks do exist. Unless the attacks are shown, the defenses will be omitted to avoid the overheads. This is currently happening for other ML threats—cryptanalytic [17] and side-channel attacks [31]—where the newly discovered vulnerabilities have resulted in applying earlier defenses (although discussions continue on how to tune defenses to minimize their costs). Our work demonstrates the capability of the scan-chain threat for the first time and motivates that the same discussions should start for scan-chain attacks.

B. Prior Work

The most recently published attacks include CRYPTO 2020 [17], USENIX 2020 [18], and ICML 2020 [19]. In CRYPTO 2020 [17], the authors use a cryptanalytic extraction method through the API using a finite difference method, and error overapproximation through mixed-integer linear programming (MILP). In USENIX 2020 [18], the authors use a query-based supervised training of substitute model for high-accuracy extraction and second-derivative based high-fidelity extraction, but the technique is restricted to only two layers and results in significant errors which need to be recovered through another phase of learning. In ICML 2020 [19], the authors propose a *hyperplane* separation method for *ReLU* networks, but results in exponential increase in error with increase in number of layers and number of neurons-per-layer. Our proposed attack, on the other hand, is correct by construction and does not introduce any errors suggested above. Since we use finer-grained scan-chain access, the accuracy is greatly improved. Additionally, we use LP formulation whose complexity is $\mathcal{O}(n.log(n))$, which is clearly easier to solve unlike MILP, which is NP-Hard.

C. Switch between modes to extract layer-wise information

The activation register scan-chain vulnerability illustrated earlier in Figure 1 can be exploited only in the chip's *debug/test mode* of operation. However, the NN classification takes place in the chip's *functional mode* of operation. Hence, it is important to understand how to carefully switch between these two modes through the appropriate timing of the multiple control signals involved. Figure 5 shows the timing of the clock (CLK), test mode (TMS), and scan enable (SE) signals. For each applied input, the NN accelerator is configured to execute the classification task using a *fast (system) clock* and subsequently the results of interest captured at the activation registers are scanned-out using a *slow (shift) clock*.

This $\{classification,\ scan-out\}$ sequence is repeated until all the activation results corresponding to the applied inputs are available. To facilitate the switch between functional mode and debug mode, the clock is gated-off (grey region) between the classification mode and scan-out mode, and the chip is switched from functional mode to debug mode by transitioning TMS from $0 \rightarrow 1$ and SE from $0 \rightarrow 1$ as shown in Figure 5. The switch between functional and debug modes also exists in the prior works on scan-chain attacks for cryptographic accelerators [11], but the timing of the clock and other control signals is unique to NN accelerators.

1) Timing of Control Signals: The NN accelerator contains a total of p PEs (refer Table I) executing in parallel, with each PE containing a multiply-and-accumulate (MAC) unit. Assume a filter-size of W_i for a neuron in layer-(i), filter-row size of f within each PE. If there were an unlimited number of PEs, each PE reduces f summands to one in $(f-1)$ clock cycles and there are $\lceil \log_f(W_i+1) \rceil$ reduction stages. Thus, the number of functional clock cycles of classification for a neuron in layer-(i) is $T_i = (f-1) * \lceil \log_f(W_i+1) \rceil$. Because the NN accelerator contains only p PEs [16], it takes more functional clock cycles:

$$T(i) = (f-1) * \sum_{m=1}^{\lceil \log_f(W_i+1) \rceil} \left\lceil \frac{(W_i+1)}{f^m.p} \right\rceil \tag{18}$$

Since we do not scan-in, in order to scan-out layer-(i), we will need to execute the classification from layer-(0) all the way till layer-(i). Hence, the total number of functional clock cycles of classification needed to capture the response to a query, for a neuron in layer-(i) is given by:

$$T_c(i) = \sum_{k=0}^{i} T(i) = (f-1) * \sum_{k=0}^{i} \sum_{m=1}^{\lceil \log_f(W_i+1) \rceil} \left\lceil \frac{(W_i+1)}{f^m.p} \right\rceil \tag{19}$$

Considering $DPFP$ representation, the number of clock cycles spent in scanning out the activation register is 64. Thus, both TMS and SE signals will be turned high during the clock gated phase (grey region) prior to scan-out and will remain high for 64 cycles of the *slow clock*, and will be turned off during the clock gated phase (grey region) post-scan-out.

VII. Conclusions and Future Work

This paper has uncovered and quantified the effectiveness of a new threat vector for model stealing from ML hardware. The proposed attack abuses the debug ports (i.e., scan-chain access) that may be active after deployment to conduct in-field tests, and it observes intermediate states of the neural network inference for efficient model extraction. Although such attacks are well-known for cryptographic circuits, they have not been explored before for ML. We have revealed that ML has unique challenges due to the number of parameters, non-injective non-linear functions, and floating-point arithmetic, which are not seen in cryptographic systems. But these challenges can be overcome with novel attack algorithms such as the ones we have proposed and with automation. The results have shown clear advantages of our proposed attack over cryptanalytic extraction both in terms of accuracy and the number of queries needed. This paper, therefore, calls for defenses against such attacks, e.g., by extending the ones built for cryptographic circuits.

To be consistent with earlier attacks and to evaluate NN achieving the best accuracy results, we used NN with floating-point weights and biases. An interesting extension of this work could be analyzing quantized networks.

References

[1] C. Li, "OpenAI's GPT-3 language model: A technical overview," 2020. [Online]. Available: https://lambdalabs.com/blog/demystifying-gpt-3/

[2] F. Tramèr et al, "Stealing Machine Learning Models via Prediction APIs," in *USENIX Security Symposium*, 2016, pp. 601–618.

[3] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 36–52.

[4] M. Juuti et al, "PRADA: Protecting Against DNN Model Stealing Attacks," in *IEEE European Symposium on Security and Privacy*, 2019, pp. 512–527.

[5] D. Lowd and C. Meek, "Adversarial learning," in *ACM KDD*, 2005, pp. 641–647.

[6] N. Papernot et al, "Practical black-box attacks against machine learning," in *AsiaCCS*, 2017, pp. 506–519.

[7] T. Gu et al, "BadNets: Evaluating Backdooring Attacks on Deep Neural Networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

[8] "Texas Instruments: IEEE Std. 1149.1 (JTAG) Testability Primer," 2016. [Online]. Available: https://www.ti.com/lit/an/ssya002c/ssya002c.pdf

[9] "Tessent Boundary Scan," 2019. [Online]. Available: https://www.mentor.com/products/silicon-yield/products/boundary-scan

[10] "1149.1-2013 - IEEE Std. for Test Access Port and Boundary-Scan," 2013. [Online]. Available: https://standards.ieee.org/standard/1149_1-2013.html

[11] B. Yang et al, "Scan based side channel attack on dedicated hardware implementations of Data Encryption Standard," in *ITC*, 2004, pp. 339–344.

[12] M. Agrawal et al, "Scan Based Side Channel Attacks on Stream Ciphers and Their Counter-Measures," in *INDOCRYPT*, 2008, pp. 226–238.

[13] K. Rosenfeld and R. Karri, "Attacks and Defenses for JTAG," *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 36–47, 2010.

[14] R. Nara et al, "Scan-based attack against elliptic curve cryptosystems," in *ASP-DAC*, 2010, pp. 407–412.

[15] J. D. Rolt et al, "A Novel Differential Scan Attack on Advanced DFT Structures," *ACM TODAES*, vol. 18, no. 4, pp. 36–47, 2013.

[16] Y. Chen et al, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE JSSC*, no. 1, pp. 127–138, 2017.

[17] N. Carlini et al, "Cryptanalytic Extraction of Neural Network Models," in *CRYPTO*, vol. 12172, 2020, pp. 189–218.

[18] M. Jagielski et al, "High accuracy and high fidelity extraction of neural networks," in *USENIX Security Symposium*, 2020, pp. 1345–1362.

[19] D. Rolnick and K. P. Körding, "Identifying Weights and Architectures of Unknown ReLU Networks," *ICML*, vol. abs/1910.00744, 2020.

[20] "ASE Tech. Holding Revenue 2006-2020," 2020. [Online]. Available: https://www.macrotrends.net/stocks/charts/ASX/ase-technology-holding/revenue/

[21] "ASE Group Test Service," 2021. [Online]. Available: https://ase.aseglobal.com/en/products/test

[22] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Smartcard*. USENIX Association, 1999.

[23] L. Alrahis, M. Yasin, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "Scansat: Unlocking obfuscated scan chains," in *ASPDAC*, 2019, p. 352–357.

[24] R. Torrance and D. James, "The state-of-the-art in semiconductor reverse engineering," in *DAC*, 2011, pp. 333–338.

[25] A. Dubey et al, "MaskedNet: A Pathway for Secure Inference against Power Side-Channel Attacks," *HOST*, 2020. [Online]. Available: https://arxiv.org/abs/1910.13063

[26] H. Yu et al, "DeepEM: Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage," in *HOST*, 2020.

[27] V. Nair et al, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *ICML*. Omnipress, 2010, pp. 807–814.

[28] S. E. Laux, "Techniques for small-signal analysis of semiconductor devices," *IEEE Transactions on Electron Devices*, vol. 32, no. 10, pp. 2028–2037, 1985.

[29] "Eyeriss Chip Simulator," 2020. [Online]. Available: https://github.com/jneless/EyerissF

[30] "The MNIST database of handwritten digits." 2020. [Online]. Available: https://www.tensorflow.org/datasets/catalog/mnist

[31] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel," in *USENIX Security Symposium*, 2019, pp. 515–532.

[32] B. Yang, K. Wu, and R. Karri, "Secure scan: A design-for-test architecture for crypto chips," *IEEE TCAD*, vol. 25, no. 10, pp. 2287–2293, 2006.

[33] D. Hely, F. Bancel, M. Flottes, and B. Rouzeyre, "Test control for secure scan designs," in *ETS*, 2005, pp. 190–195.

[34] S. S. Ali et al, "Novel test-mode-only scan attack and countermeasure for compression-based scan architectures," *IEEE TCAD*, vol. 34, no. 5, pp. 808–821, 2015.