UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Side-Channel Attacks against Neural Network Hardware Accelerators

Seitenkanalangriffe auf Hardwarebeschleuniger für Neuronale Netze

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Alexander Treff

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Okan Seker
Thore Tiemann

Lübeck, den 22. Juli 2021

Abstract

Machine learning (ML) becomes more and more important to solve a variety of problems, including autonomous driving and medical applications. Since a ML model contains valuable information not necessarily known to third parties, the model itself as well as all the training samples are considered intellectual property often worth millions of dollars. With the rise of Internet-of-Things (IoT) devices, ML moves closer to the end user. This opens new attack avenues to an adversary with physical access to the device running the corresponding ML model. Several companies offer special ML acceleration chips for power-efficient usage in edge devices. The ML accelerator chip that we investigate in this thesis is Intel's Neural Compute Stick 2; a plug and play development kit for embeddded ML acceleration. We investigate how an adversary with physical access to the edge device can learn internals about the deployed ML model through different side-channels, including the USB communication as well as power consumption and electromagnetic emanations. By eavesdropping the USB communication during deployment of a model, an adversary can perfectly recover the model. Moreover, by measuring the power consumption and EM emanations of the accelerator device during model inference, we successfully reconstruct the model's structure with only a single inference request using Simple Power Analysis. Both attacks do not rely on expensive equipment and are thus easily applicable even outside of specialized labs or institutions, making these attacks a serious threat.

Kurzfassung

Maschinelles Lernen (ML) nimmt eine immer wichtigere Rolle ein, um eine Vielzahl an Problemen zu lösen. Dazu gehören beispielsweise Autonomes Fahren oder medizinische Anwendungen. Ein ML-Modell enthält wertvolle Informationen, die Dritten nicht notwendigerweise zugänglich sind. Daher sollten sowohl das Modell selbst als auch die Trainingsdaten, die für das Modell verwendet wurden, als geistiges Eigentum eingestuft werden. Dessen Wert beträgt leicht mehrere Millionen Dollar. Mit dem weiteren Aufkommen von Internet-der-Dinge-Geräten (engl. Internet-of-Things (IoT)) rückt auch ML näher an den Endanwender heran. Einem Angreifer mit physischem Zugriff auf das Gerät, auf welchem das entsprechende ML-Modell ausgeführt wird, werden so neue Angriffsmöglichkeiten eröffnet. Einige Unternehmen bieten inzwischen spezielle ML-Beschleunigerkomponenten an, die insbesondere auf ihren Stromverbrauch optimiert wurden und sich somit für sogenannte edge devices eignen. Die ML-Beschleunigerkomponente, die wir in dieser Arbeit untersuchen ist der Neural Compute Stick 2 von Intel. Wir untersuchen inwiefern ein Angreifer mit physischem Zugriff auf das Gerät geheime Informationen über das aktuell ausgeführte ML-Modell erhalten kann. Dazu untersuchen wir verschiedene Seitenkanäle: einerseits die USB-Kommunikation, andererseits den Stromverbrauch und elektromagnetische Strahlung des Geräts während des Betriebs. Ein Angreifer, der die USB-Kommunikation des Geräts belauschen kann, während ein Modell bereitgestellt wird, ist in der Lage, dieses Modell perfekt zu rekonstruieren. Mittels Simple Power Analysis gelingt es uns zudem, die Struktur des ML-Modells zu rekonstruieren, indem wir den Stromverbrauch und die elektromagnetische Strahlung des Geräts während der Inferenzanfragen aufzeichnen. Dazu genügt bereits eine einzige Anfrage. Beide Angriffe erfordern keine teuren Geräte und sind somit auch außerhalb von Laboren oder Institutionen, die sich auf solche Angriffe spezialisiert haben durchführbar, sodass diese Angriffe eine reale Gefahr darstellen.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 22. Juli 2021

Acknowledgements

First of all, I want to thank Okan Seker, Thore Tiemann and Prof. Dr. Thomas Eisenbarth for their excellent supervision and continuous support while writing and working on this thesis. Additionally, I want to thank Prof. Dr. Oliver Stecklina from the TH Lübeck for allowing me to use the positioning table at the *Lab for secure hardware and software development*. Last but not least, I want to thank Sven Dreier and Ulrich Behrje from the Institute of Computer Engineering (ITI) for helping me out with different hardware tasks.

Contents

List of Tables

List of Figures

1 Introduction

Deep learning based on neural networks is ubiquitous nowadays and plays an increasingly important economical role. A broad range of applications, e.g. image processing [Tea17], speech recognition [Sch19], pattern recognition for medical diagnosis, autonomous driving or text auto-completion [Che+19; Lam18] take advantage of deep learning techniques. For all of these applications, individual training samples are supplied as inputs to the deep learning algorithm which then trains and outputs a *model* that can be used for future queries.

Neural networks are rapidly becoming more advanced, and thereby are able to solve more complex problems with higher accuracy. This is accomplished by intensive research progress accompanied by increasingly available (and affordable) computational power, e.g. by utilization of GPUs [Che+14; RMN09] or dedicated hardware such as specialized ASICs (Application-specific integrated circuits) like Google's Tensor Processing Units (TPUs) [Jou+17]. For increasingly complex tasks, huge amounts of data are necessary during training, yielding models that are highly accurate, but also expensive to generate and train.

It takes lots of inputs and training iterations to obtain a model with the desired accuracy. This takes time and costs money to acquire or rent the required hardware as well as the costs incurred by the power consumption. Training data might also not always be publicly available – a suitable data set then needs to be bought from external suppliers. Furthermore, the inputs used throughout the training phase may contain sensitive information, e.g., medical diagnostics for specific individuals, which can potentially be extracted from the models [Zha+20], requiring good protection for Machine learning (ML) models. Both the training samples as well as the parameters of the model that have been learned through training should therefore be treated as sensitive data or intellectual property. When the network has achieved the desired accuracy, it can be deployed to the target to run *inference tasks* (e.g., classification or prediction queries).

Several Machine Learning as a Service (MLaaS) providers give query access to their pre-trained models via their API over the internet. The customer typically pays for each query that is sent to the cloud and receives the result for further computations. However, extra care has to be taken in order to protect the models deployed on products handed to end-users as attackers may attempt to steal the model to benefit from it while avoiding the high training and query costs. This threat is real for MLaaS providers [Tra+16; WG18] and

Figure 1.1: The NCS 2.

companies selling machine learning enabled products alike.

On the other hand, modern machine learning accelerators become more and more efficient. This allows for including chips optimized for running inference tasks directly into the machine learning enabled product that is shipped to the end-user. Running inference on edge devices instead of sending queries to remote servers has several benefits. First, data can be used where it is created. This helps increasing the privacy of the classified data. Second, less bandwidth is necessary to serve queries as inference accelerators are distributed to the edge devices. Especially industries implementing machine learning features into their products while producing thousands of products like the automotive industry or the Internet-of-Things (IoT) might otherwise see the bandwidth to be a bottleneck.

Since the inference tasks typically are computationally expensive as well, low power devices such as the Raspberry Pi might not be able to fulfill strict time constraints for inference tasks. To fix this shortcoming, hardware accelerators can be used to increase the throughput of neural network inferencing in these special low-power environments. The model is loaded onto the accelerator where the inference task can be performed much faster.

In order to allow more powerful machine learning to happen directly in IoT or edge devices, special-purpose hardware accelerators are being deployed. These accelerators increase throughput of neural network inference in such special low-power environments. One of these low-power inferencing hardware accelerators is Intel's *Neural Compute Stick 2 (NCS 2)* [Inta] that we analyze in this work. The NCS 2 is depicted in Figure 1.1.

The NCS 2 is a hardware accelerator for neural network inference tasks especially in low-power, embedded environments. It is entirely powered via USB, therefore no external power supply is needed. The Intel Movidius Myriad X Vision Processing Unit (VPU) is built onto the stick. Possible use cases mentioned by the manufacturer include *smart cameras, drones and smart-home devices* [Intb; Intc].

1.1 Contribution of this Thesis

As all IoT devices, the NCS 2 enables physical access of an adversary. Therefore, physical attacks such as Side-Channel Analysis (SCA) cannot be ignored.
Our aim is to investigate to what extent an attacker is able to exfiltrate possibly confidential information regarding the currently deployed model on a NCS 2 through different variants of SCA. We formulate two research questions that we answer throughout the thesis.

Research Question 1: *Does the NCS 2 leak secret information about the internal structure of the currently deployed model through side-channels?*
To answer the first research question, we analyze different side-channels such as the USB data communication, the power consumption, EM emanations and timing behaviour [Koc96; KJJ99; QS01] of the device during model deployment and inference.

Research Question 2: *Is it possible to reconstruct the entire model from side-channel leakages, if there are any?*
The second research question builds on top of the first research question. The attack complexity increases since the leakage needs to be meaningful enough to recover not only the structure but also single parameters of the model.

1.2 Thesis Structure

In Chapter 2, we give an introduction to both side-channel attacks and neural networks as well as specially crafted hardware accelerators for the latter. Chapter 3 presents an overview over the current state of the art about attacks against neural networks as well as side-channel analysis of hardware devices that are running those. The first attack that we present in Chapter 4 is accomplished by eavesdropping the USB communication between the host and the NCS 2. Then in Chapter 5 we describe how observing the power consumption of the NCS 2 enables an adversary to learn the model's strucure. Finally, we give a conclusion and propose ideas for future work including possible countermeasures in Chapter 6.

2 Background

This chapter first provides an introduction to deep neural networks and side-channel attacks in general. Afterwards, we give an overview of neural network hardware accelerators and describe how the OpenVINO (Open Visual Inference and Neural Optimization toolkit) framework is being used to operate with the target device. Lastly, we introduce the target device itself.

2.1 Neural Networks and Their Components

In this section we begin by describing simple neural networks and their components. The idea of neural networks builds upon the concepts that McCulloch and Pitts [MP43] first described back in 1943 already. Afterwards, we introduce several concepts that evolved over time that modern neural networks build upon.
We refer to Schmidhuber's thorough overview of the development of neural networks for further reference [Sch15].

2.1.1 Introduction

Research of neural networks spreads into a variety of directions. Popular research directions include the optimization of the learning process, by designing entirely new concepts, finding optimal network layouts for existing concepts or optimizing learning strategies. Other researchers focus on the *explainability* or *privacy* aspects of learning or try to improve their tolerance against faulty inputs.
The first step is to *train* a network using *training samples* to fine-tune the *parameters* of the network.
For the rest of this thesis however, we explicitly do **not** focus on the beforementioned aspects since the device under test (DUT) we examine (NCS 2) is specifically crafted for what is called *inference*. Therefore we assume that a network has already been trained on a dataset, reached the desired accuracy and is ready for deployment. Thus we do not go into detail about concepts such as *overfitting*, *regularization*, *dropout* etc. which are important in the learning phase and just name them here shortly for sake of completeness.

Tensors In the domain of neural networks one often comes across the term *tensor*. A tensor is a mathematical object that generalizes the concept of scalars, vectors and matrices

to higher-dimensional structures. The equivalent from a computer scientist's perspective is a multi-dimensional array. Throughout this section, we continue to use the terms *vector* and *matrix* for sake of simplicity. When considering networks with higher-dimensional inputs or internal structures, the corresponding tensor operations (tensor addition, tensor product etc.) are meant instead. We don't go more into detail since we are only interested in *reconstructing* the network – not in understanding the actual computation.

2.1.2 Network Architecture

We describe neural networks using a top-to-bottom approach, starting with a high level introduction and disassemble the introduced concepts into more detailed views of their concepts.
Neural networks are organized in *layers*. Layers are classified into three categories: *input layers*, *output layers* and *hidden layers*. The *input layer* describes the input to the network, whereas the *output layer* represents the output of the network. All additional layers are *hidden layers*. Hidden layers are located between the input and output layer. Several types of layers exist – the most common types of layers, namely fully-connected layers and convolutional layers, will be introduced in the upcoming paragraphs. Each layer has a set of *hyperparameters*. Hyperparameters depend on the type of the layer. Depending on the layer type and its hyperparameters, each layer consists of a quite large number of *parameters*. These *parameters* are the values that are actually learned throughout training whereas the *hyperparameters* are guessed or tried out beforehand. A network is fully described by its *structure* (i.e. layers), their *hyperparameters* and their *parameters*.

2.1.3 Multi-Layer Perceptron

The simplest form of neural networks is the Multi-layer perceptron (MLP). MLPs consist of an input and an output layer as well as at least one *hidden* layer.
MLPs are fully-connected networks, i.e., each neuron from one layer is connected to each neuron from the next layer. Therefore the number of connections within the network (and thus the number of parameters) rapidly increases for a larger number of layers as well as larger inputs.
A simple, fully-connected MLP with four inputs, a hidden layer containing five neurons and one output is depicted in Figure 2.1.
Each layer consists of a specific number of *neurons*. The number of neurons in the input layer is given by the size of the input. Similarly, the number of neurons in the output layer determines the number of outputs. A layer is *fully-connected* (also called *dense*) if each neuron of the previous layer is connected to each neuron from the current layer. Values are

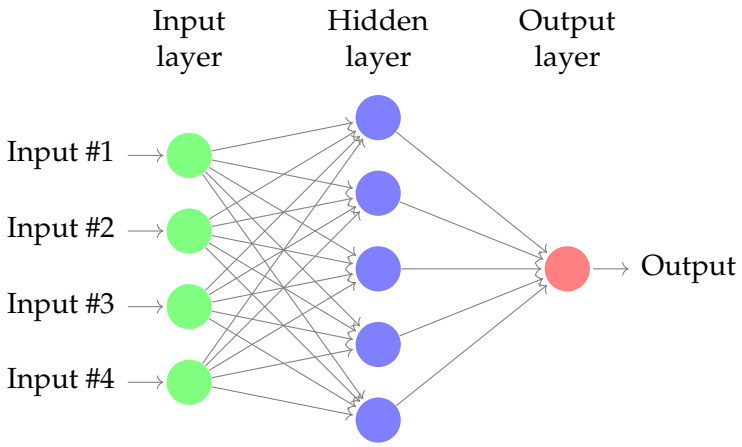


Figure 2.1: A Multi-layer perceptron with one hidden layer.

being propagated throughout the network, beginning with the input values.

For a fully-connected layer, the number of neurons is one of the hyperparameters of that layer.

For example, consider a dataset that contains scans of handwritten digits, each being normalized to a dimension of 28x28 pixels. A network trained on this data set therefore starts with an input layer consisting of 28x28=784 neurons – one neuron for each pixel. Since the aim is to classify the written digit, the network's output layer consists of 10 neurons – one neuron for each digit. We will have a closer look on such a dataset in Subsection 2.1.6.

Weights and Biases Each connection between two neurons within the network has a *weight*. This weight determines how much impact the information that comes from the source neuron should have for the target neuron. The value of a neuron is computed by summing up all the incoming values multiplied by their corresponding weights.

Additionally, each neuron has a *bias*. The bias is added to the sum of weighted inputs and adds more flexibility to the network. Because it implicitly adds another input dimension, it allows a neuron to compute a result that is different from zero even if all inputs are zero.

The calculation of a single neuron's output is illustrated in Figure 2.2.

Operations in a neural network consisting of fully-connected layers can therefore be expressed as matrix-vector multiplications to compute the results of all neurons of a

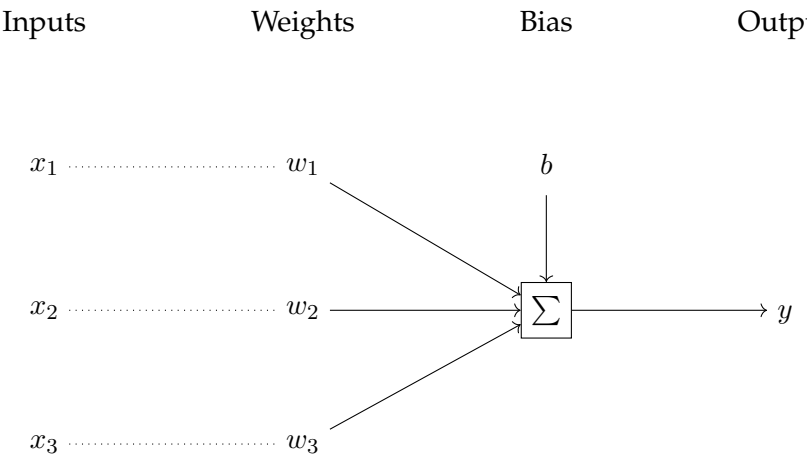


Figure 2.2: A single neuron with three inputs. The output value is $x_1w_1 + x_2w_2 + x_3w_3 + b$.

specific layer:

$$\mathbf{Y} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad \text{where } \mathbf{Y} \text{ is the output vector, } \mathbf{W} \text{ is the weight matrix, } \mathbf{x} \text{ is the input vector and } \mathbf{b} \text{ is the bias vector.} \tag{2.1}$$

2.1.4 Activation Functions

To capture non-linear dependencies in the training data, non-linearity needs to be introduced into the network – otherwise the network can only learn linear dependencies. Since the calculation of a neuron is only an affine operation (or even a linear operation when the bias is set to zero), the concept of neurons does not fulfill this property on its own. To introduce non-linearity, an *activation function* is applied to the output of a neuron.

Different activation functions have been proposed and actively used in the context of neural networks. We begin by explaining the three most popular activation functions *sigmoid*, *tanh*, and *ReLU (Rectified Linear Unit)*. These three activation functions have in common that they operate on each element of the input vector independently. Afterwards, we define the *SoftMax* activation function whose output value depends on the whole input vector.

The activation function that is being applied is considered as one of a layer's hyperparameters.

Sigmoid

The sigmoid activation function σ is illustrated in Figure 2.3a. Sigmoid is defined as:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2.2}$$

The output of σ is always in the range $(0, 1)$.

Hyperbolic Tangent

The hyperbolic tangent $\tanh$ is depicted in Figure 2.3b. There are several ways to define the hyperbolic tangent. One possible way is to define it via the sigmoid function:

$$\tanh(x) = 2\sigma(2x) - 1 \tag{2.3}$$

The output of $\tanh$ is always in the range $(-1, 1)$.

ReLU

The ReLU activation function is shown in Figure 2.3c. ReLU is defined as:

$$ReLU(x) = \max(0, x) \tag{2.4}$$

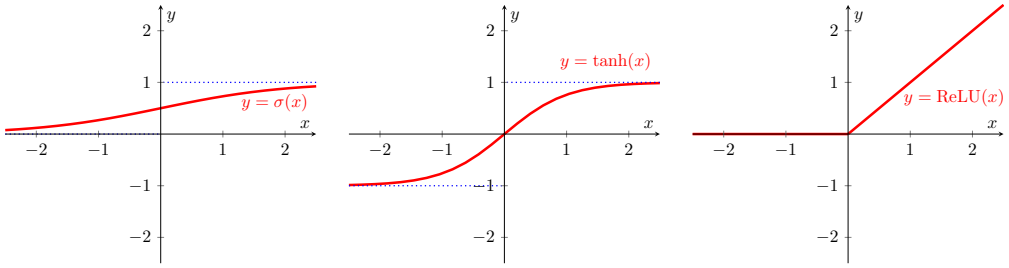
If the input is smaller than zero it will be set to zero. The output of ReLU is always non-negative, but in contrast to $\tanh$ and σ there is no upper bound. Due to its simplicity it gained popularity since training times drastically decrease when this activation function is being used.

SoftMax

The *SoftMax* activation function is a special activation function. It is often the last activation function of a network. Its purpose is to normalize the output values to be within the range $[0, 1]$ and is defined as

$$\sigma(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K \tag{2.5}$$

After applying softmax to an input vector, the elements of the output vector sum up to one, thus the result fulfills the requirements of a probability distribution. SoftMax is usually used after the last dense layer to obtain a distribution-like output as described beforehand.



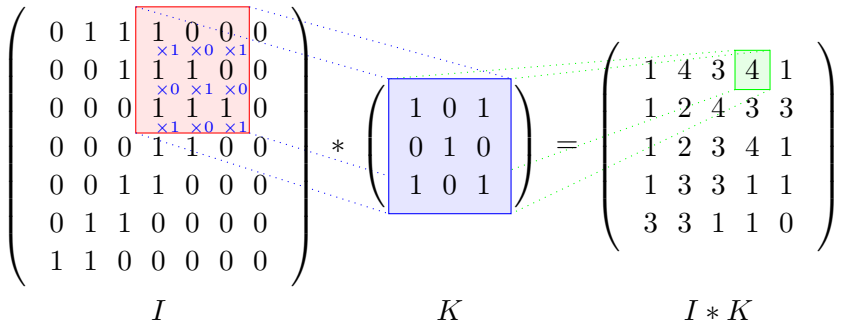
(a) Sigmoid function (b) Tanh function (c) ReLU function

Figure 2.3: Popular activation functions in neural networks.

2.1.5 Convolutional Neural Networks

Convolutional neural networks (CNNs) are more advanced networks that contain *convolutional* layers. The ideas behind CNNs date back into 1980 when Fukushima published his paper about *Neocognitron* [Fuk80]. LeCun et al. further developed the concept in their seminal papers [LeC+89; LBBH98].
CNNs contain several convolutional layers to extract the *features* (e.g., edges, shapes etc.) of the input. The last layer(s) are then fully-connected layers that perform the actual classification on the latent (hidden) representation generated by the convolutional layers. Hyperparameters of convolutional layers are the *kernel size*, *stride*, *padding*, the *activation function* and possibly even more.
Convolutional layers are advantageous when processing higher-dimensional data in which MLPs would have too many connections. They make use of two important properties: *locality* and *spatial invariance*. *Spatial invariance* means that the CNN recognizes features regardless of their exact position within the input. *Locality* on the other hand described the fact that especially for image recognition, it is the case that pixels are correlated to pixels very close to them whereas they are usually not correlated to pixels that are far away. Thus, in contrast to fully-connected feed-forward networks, CNNs have far less trainable parameters since neurons that correspond to pixels far away are simply not connected, making them easier to train and deploy for higher input dimensions.

Convolution Operation Convolution is computed by first element-wise multiplying the input window and the convolution kernel. A convolution *kernel* (or *filter*, *feature map*) is simply a matrix of weights that is being shifted across the input matrix during convolution. Strictly speaking this operation is not a convolution, but a *cross-correlation* instead, since for a convolution the kernel would be flipped beforehand. Afterwards, these values are being summed up, yielding the final output value at the corresponding index of the output matrix.



(2.6)

An example for a convolution is given in Equation 2.6 where a 7×7 input matrix is convolved using a 3×3 kernel, yielding an output matrix of size 5×5. When considering a kernel that has the same size as the input with no padding and a stride value of 1, this operation is equivalent to a fully-connected layer. Hence, convolutional layers are a generalized version of fully-connected layers.

Stride As seen above, convolution results in a small dimensionality reduction. Sometimes, a more aggressive dimensionality reduction is desired, for example when the input is too large and we wish to operate on a lower resolution for performance reasons. In these cases, increasing the *stride* helps. The stride is the step size during convolution. By default, the stride is set to 1. Then during convolution, the kernel window is moved by one element per step. By increasing the stride, the output's dimension becomes smaller and the number of convolutions decreases.

Padding When using convolution kernels with higher dimensions than 1×1, the resulting output has a lower dimension than the input. This directly results in information loss on the boundaries of the original input. The most often used technique to mitigate the information loss is *padding*. When using padding, the input is augmented by adding additional rows and columns consisting of zeros around the input matrix. Depending on the size of the padding (and the size of the kernel), there will be less dimension-reduction, no dimension-reduction, or the dimension even grows a bit.

Now that all popular techniques around convolution have been explained, we give a

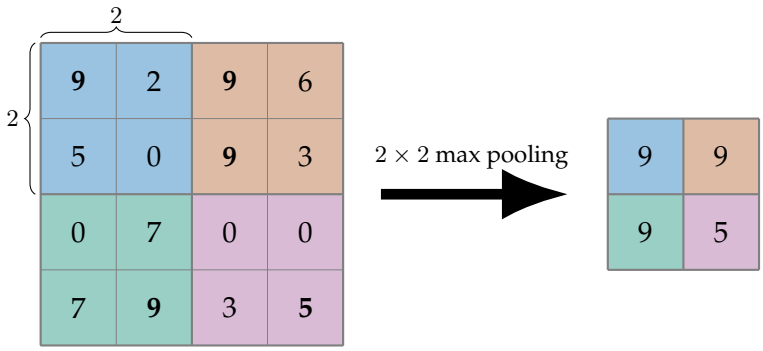


Figure 2.4: Application of 2x2 MaxPooling to a 4x4-matrix, resulting in an output of dimensions 2x2. The input is tiled into 2x2 pools from which the maximum value is taken as the result.

formula to easily compute the output dimensions after convolution:

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor \quad \text{where}$$
$$n_h, n_w \text{ input}$$
$$k_h, k_w \text{ kernel} \tag{2.7}$$
$$p_h, p_w \text{ padding}$$
$$s_h, s_w \text{ stride}$$

This formula applies to a 2D-convolution. For higher-dimensional convolution, the formula needs to be adjusted accordingly.

Pooling

Pooling, especially MaxPooling is used to reduce the latent dimensions in hidden layers. Given an internal representation of dimensions $m \times n$, $(i \times j)$-pooling reduces the dimensions to $(\lfloor \frac{m}{i} \rfloor \times \lfloor \frac{n}{j} \rfloor)$ by considering $i \times j$ *windows* and (in the case of MaxPooling) picking the maximum value within this window.

An example is given in Figure 2.4 where (2×2)-maxpooling is applied to an input matrix of dimensions 4×4 yielding an output of dimensions 2×2. The maximum values of each window are highlighted. Pooling can also be combined with the beforementioned *padding* and *stride* techniques to adjust the output dimensions.

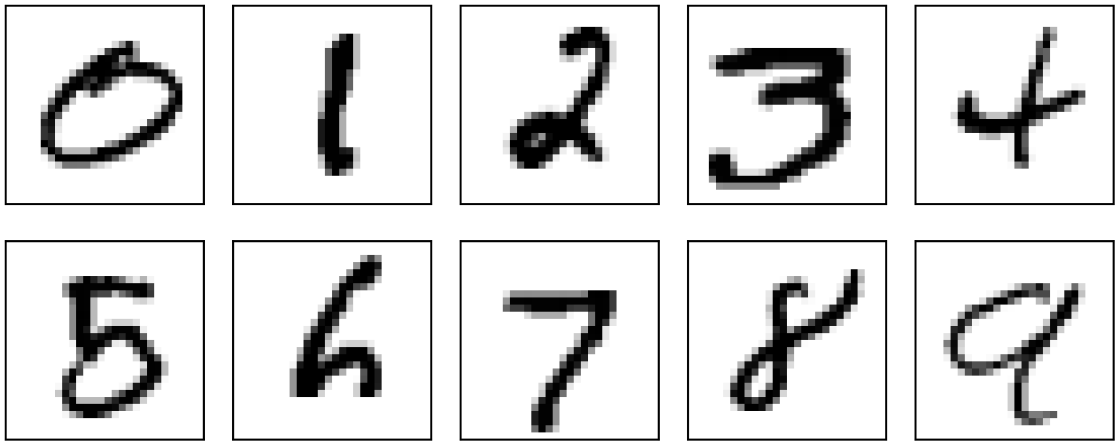


Figure 2.5: A few samples drawn randomly from the MNIST dataset. Some of them are easy to read, others are ambiguous.

2.1.6 The MNIST Dataset

We describe the popular MNIST dataset and a corresponding neural network architecture to achieve good accuracy results for it. The MNIST dataset[1] [LBBH98] contains a total of 70.000 handwritten digits which are split into a training set consisting of 60.000 images and a test set consisting of 10.000 images. Each digit is represented as a 28×28 pixels grayscale image. Figure 2.5 shows a few samples randomly drawn from the dataset. MNIST is one of the most popular datasets used in machine learning nowadays. It is well suited for beginners due to its small sample size and an easily to achieve high accuracy even with simple networks that take little training time.
We now give a short example of a simple CNN that has been trained on the MNIST dataset. Figure 2.6 shows the general network layout. The network contains two convolutional layers to extract the features, each followed by a MaxPooling layer to reduce the latent dimensions.
After flattening the internal representation, two fully-connected layers perform the actual classification. The output is then fed into a SoftMax layer to obtain a normalized, distribution-like output.

2.2 Side-Channel Attacks

Physical attacks are a well-known threat to cryptosystems especially if the adversary has (but not necessary needs) physical access to the implementation. There exist two main

[1] More detailed information about MNIST and research results are available at http://yann.lecun.com/exdb/mnist/

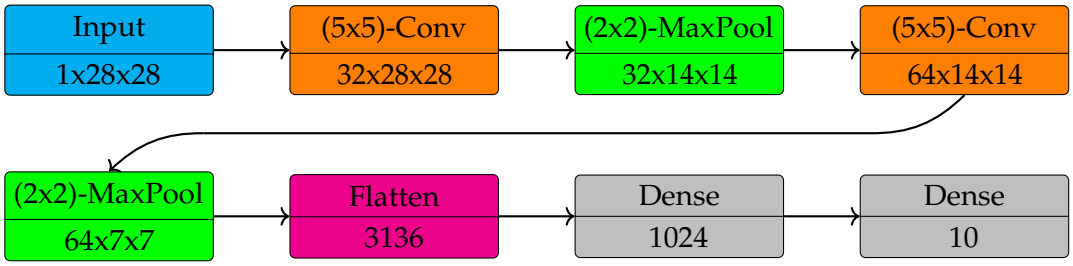


Figure 2.6: A simple MNIST classifier. The lower part of each layer describes its dimensions: Features × Width × Height. The input has one feature per pixel: its grayscale value. Over time, the feature dimension grows, while the width and height shrinks by combining convolutional and max-pooling layers with kernels of corresponding size. Finally, the latent representation is flattened into one dimension and two dense layers perform the actual classification. Softmax is applied after the last dense layer to obtain a distribution-like output.

classes of attacks: active *fault injection attacks* and passive *side-channel attacks*. While fault attacks manipulate the internal state of a cryptographic implementation, side-channel attacks exploit the physical information given by the device on which the implementation of an algorithm is running. The physical information can be exemplified as timing information [Koc96], cache behavior [TOS10; YF14], emitted sound [GST14], power consumption [KJJ99], or electromagnetic emanation [GMO01]. In this work we focus on side-channel analysis and in the following we focus on one of the main attacks: Simple Power Analysis (SPA).

2.2.1 Timing Attacks

Timing attacks were first introduced by Kocher in 1996 [Koc96]. The leakage in this case is the differing timing behavior, depending on which operations are executed. Often, different operations take different amounts of time and an adversary can take advantage of this fact. For example, consider the different activation functions described in Subsection 2.1.4. Comparing the computation of tanh and ReLU, one observes that tanh is a lot more complex than ReLU. Two expensive operations, namely the exponentation followed by a division are needed to compute the result of the sigmoid function which in turn is used to compute the result of tanh. On the other hand, for ReLU, depending on the sign returning zero or the input value itself is sufficient. This results in different timing characteristics that might leak information about the function that has been used.

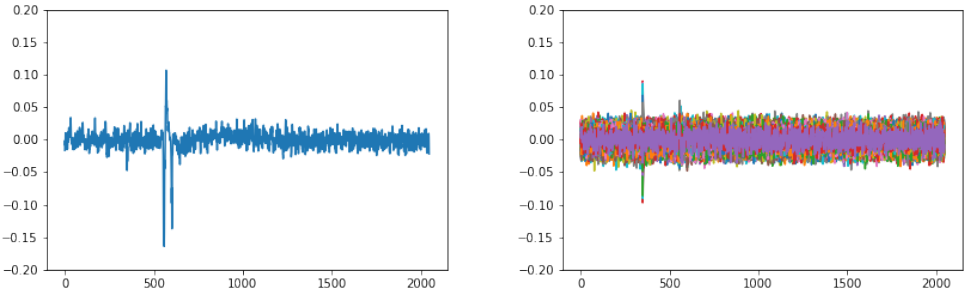
2.2.2 Simple Power/EM Analysis (SPA/SEMA)

SPA has been introduced by Kocher, Jaffe, and Jun [KJJ99]. The target's power consumption is measured during its operation. When targeting the EM emanations instead, this attack is called SEMA. The idea of this attack is that different operations running on the DUT result in a slightly differing power consumption/EM emanation. SPA can reveal information about the current operation, e.g. a repeating pattern such as encryption rounds or even sensitive information such as key material. One popular example for the success of SPA is the key recovery of unprotected RSA implementations. The side-channel trace changes with respect to the longer multiplication which implies the secret key bit is 1 or the faster squaring which implies the secret key bit is 0 [Fou+06]. Since the power consumption differs for those two cases, the adversary is able to distinguish the performed operation and can directly infer the bit values of the private key.
The same example (neural network activation functions) that has been given in the context of timing attacks holds for SPA as well. In fact, Takatoi et al. [TSSL20] even suggest to use SEMA since it withstands attempts to make the execution time of all activation functions constant.

2.2.3 Differential Power/EM Analysis (DPA/DEMA)

Differential Power Analysis (DPA) has been proposed by Kocher, Jaffe, and Jun [KJJ99] and is another commonly used side-channel attack. It is a more complex attack and usually requires more power/EM traces. Then, statistical methods are employed to exploit dependencies between the side channel information and the processed data to learn about secret values.
In most cases, the leakage imposed by the correlation of the power consumption and the processing of internal data of the DUT is very small and hides within the background noise. In many cases it is still possible to gain information about internal data structures (such as a secret key) by applying DPA.
DPA is implemented by selecting an intermediate variable that depends on a secret variable (such as *secret key*) and a known value (such as *plaintext*). Then, the adversary selects a function that models the power trace such as Hamming weight [BCO04] or a single bit [MOP07] to be used in the statistical analysis phase. Therefore, the adversary can model the power consumption for each key candidate[2].
The wrong key guesses result in wrong power models and thus the statistical analysis results do not show any significant correlation peak. However the correct key which results in the correct power model, and therefore the analysis results in an observable peak (if no

[2]The key space is small enough to process the analysis and generally it ranges from 2^1 to 2^{16}.



(a) Correct key guess.

(b) Wrong key guesses.

Figure 2.7: DPA against an unprotected AES implementation. A significant peak can be observed between samples 500 and 600 for the correct key guess. The wrong key guesses show no such peak between samples 500 and 600.

countermeasures are present).
One popular example when performing a DPA against AES is to target the output of the S-box in the first round. We visualize a simple DPA against an unprotected AES implementation in Figure 2.7. The upper plot shows the results for the correct key guess where a significant peak can be observed. The lower plot shows all results for incorrect key guesses. An attacker recovers one secret key byte at a time with this approach, hence the attack has to be performed 16 times for AES-128 to recover the entire key.

2.3 Combining SCA and NN

In Sections 2.1 and 2.2, we introduced neural networks and SCA as distinct topics. More recently, these two topics have been composed in a common context in (at least) two different ways:

1. Improving SCA by using ML. The first combination can be seen as a continuation of template attacks [CRR03] since the usage of neural networks provide several advantages in modern SCA. The advantage of this approach is that numerous preprocessing steps such as alignment of traces is not necessarily needed anymore. Instead, the network learns how to deal with misalignments and corrects them automatically during inference.

2. Exploiting ML by using SCA. For the second approach, SCA is utilized to recover internal information of a ML model. In this case, analogously to recovering the secret key of an encryption algorithm, the aim is to recover internal information of a ML model, such as weights.

We focus on the second approach in this thesis.

2.4 Neural Network Hardware Accelerators

Different types of hardware accelerators for neural network inference tasks exist. In this section we describe different neural network hardware accelerators, namely our main target NCS 2 as well as two of its competitors.

2.4.1 Intel Neural Compute Stick 2

The target device that we analyze is Intel's *Neural Compute Stick 2* (NCS 2, codename MYRIAD), a rapid development and testing board. It is a neural network inferencing hardware accelerator that is passively cooled via its casing and entirely powered via USB so no external power supply is needed. It is run by a Myriad X VPU [Corb] that is clocked at 700MHz. The VPU has 16 Streaming Hybrid Architecture Vector Engine (SHAVE) cores as well as 4GB memory together with an additional CMX (Connection MatriX) memory that is shared between all cores. The Myriad X is also used in other products, e.g., DepthAI [Lux], AAEON [AAE], or the OpenCV AI Kit [Ope]. Additionally, it is also available as a PCIe compute module.
One important thing to note is that the NCS 2 operates on 16-bit floating point numbers for performance reasons. Since training is usually done with 32-bit floating point numbers the resulting models have to be converted from 32-bit to 16-bit floating point numbers. By doing so, the exponent bits are reduced from 8 to 5 bits and the mantissa is reduced from 23 to 10 bits as illustrated in Figure 2.8.
An information loss is therefore unavoidable when converting the original model with its 32-bit weights to the optimized model containing just 16-bit weights. However, even though two bytes per number are lost, the accuracy of the model stays almost the same because the impact of the last few digits on the resulting classification usually is rather small. The NCS 2 automatically downsamples 32-bit inputs to 16-bit and upsamples 16-bit results to 32-bit before sending them back to the host.
The performance of the predecessor of the Myriad X, the Myriad 2, was analyzed by Rivas-Gomez et al. [Riv+18]. Also, it was used in DJI drones [Glo]. Antonini et al. [Ant+19] compared the NCS2 and other accelerators that are designed to be used in edge devices like the Google Coral or Nvidia Jetson Nano.

Operation Details

When attaching the stick to the host computer, it shows up as USB ID `03e7:2485` (Movidius Ltd., Movidius MyriadX), running at USB 2.0 speed (480 MB/s). The vendor

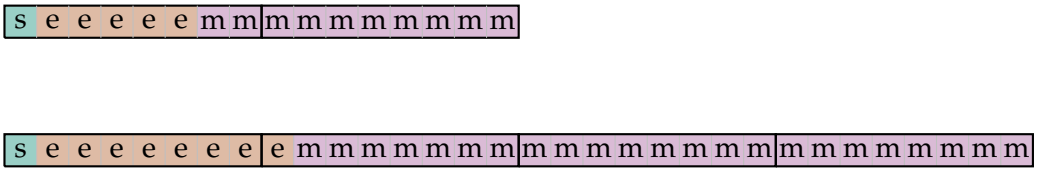


Figure 2.8: Comparison of 16-bit and 32-bit IEEE 754 floating points. 32-bit IEEE 754 floats use 8 exponent and 23 mantissa bits, whereas 16-bit IEEE 754 floats use only 5 exponent and 10 mantissa bits.

ID `03e7` belongs to Intel and `2485` is Intel's product ID for the Myriad X. To run inference tasks on the NCS 2, the device needs to be booted using a specific firmware. This firmware is loaded automatically by the OpenVINO toolkit. It is located in the directory `inference-engine/temp/vpu/firmware/usb-ma2x8x/mvnc` within the `openvino` directory. After flashing the firmware, the stick "disconnects" and show up again as `03e7:f63b` (Intel Corporation, VSC Loopback Device, different product ID) and now supports full USB 3.0 speed. When all inference tasks have finished and the script is terminated, OpenVINO shuts down the stick and it returns to its USB 2.0 state.
The NCS 2 is capable of running several networks simulaneously. Additionally, OpenVINO allows stacking several NCS 2 devices to improve performance.

Movidius Firmware The file `usb-ma2x8x.mvcmd`[3] (approx. 2 MB) that is flashed to the NCS 2 when waking it up from its idle state is a firmware file. Running `strings usb-ma2x8x.mvcmd` gives hints that the firmware is based on RTEMS [Pro]:

```
status == RTEMS_SUCCESSFUL
Starting RTEMS Shell
/home/bmanciu/Work/MovidiusRTEMS-master/MovidiusRTEMS...
rtems-5.0.0 (MYRIAD2/w/FPU/ma2x8x)
```

2.4.2 Competing Edge Accelerators

Intel's NCS 2 is not the only ML accelerator for edge devices. We found two competitors that are built for similar tasks as the NCS 2: Google's *Coral* [Goo] and Nvidia's *Jetson Nano* [NVI]. Bangash published a blogpost [Ban] where he compared the first generation Intel Neural Compute Stick against the two competitors. Please note that there is a huge performance improvement from the original Neural Compute Stick to the NCS 2.
The Google Coral is available both as a development board and as a dedicated ML co-processor, making it similar to the NCS 2. The Jetson Nano by NVIDIA differs from the

[3] SHA256 sum: `87cac2fb3a750f490af583f0c9acc6700abe11ce1fb06fa04a55d1f4e324d5aa`

NCS 2 and Google's Coral as it already ships as a whole board including flash storage and peripheral access ports such as USB, HDMI etc.

2.5 The OpenVINO toolkit

The OpenVINO toolkit is an open-source deep learning framework by Intel for different targets, such as CPUs, GPUs, Field-Programmable Gate Array (FPGA)s, or the NCS 2, as well as heterogenous setups (e.g., CPU+GPU). In this thesis we focus on the NCS 2 as our main target. OpenVINO consists of three parts:

- Model Optimizer (MO): the MO is a collection of scripts to convert and optimize existing neural networks from a variety of different frameworks such as TensorFlow [Mar+15], Caffe [Jia+14], MXNet [Che+15], Kaldi [Pov+11] to OpenVINO's internal IR. It also directly supports models in Open Neural Network eXchange format (ONNX) [BLZ+19] format. We do not go into detail on how to convert a model to IR since it is out of scope for this thesis.
- Inference Engine (IE): the IE is a set of libraries to load the converted network onto the target device and do the actual inference. This is the part of OpenVINO that we have the most to do with directly.
- nGraph: nGraph is the underlying framework that is developed and used by Intel to enable framework-agnostic deep learning tasks (both training and inference) on several hardware architectures [Cyp+18]). The IE uses nGraph internally.

The IR can either be modified through nGraph function calls (which is the official, supported way) or through directly modifiying the resulting IR (consisting of a XML+BIN file pair).

2.5.1 Loading the IR onto the Target Device

After converting the original network to OpenVINO's IR, the network can be loaded onto the target, i.e. the NCS 2 by using the OpenVINO Inference Engine. The IE supports both C++ and Python bindings – for our experiments, we stick to the Python bindings. We refer the interested reader to Appendix B for sample code on how to deploy a model to the NCS 2.

Security The security section of the OpenVINO documentation [Cor21] is rather scarce and only suggests to encrypt the model when transferring it between devices and to decrypt it just before loading it to the target device. There is no option to explicitly load an encrypted model or increase the security level in any way.

```
<layer id="11" name="conv2/Conv2D/Transpose522_const" type="Const"
↪  version="opset1">
    <data element_type="f16" offset="1728" shape="64,32,5,5" size="102400"/>
    <output>
        <port id="1" precision="FP16">
            <dim>64</dim>
            <dim>32</dim>
            <dim>5</dim>
            <dim>5</dim>
        </port>
    </output>
</layer>
```

Figure 2.9: Description of a convolution kernel for a model in OpenVINO IR.

The OpenVINO IR For our tests we use a model that is pre-trained and converted to OpenVINO IR format. The OpenVINO IR consists of an XML file and a BIN file. The BIN file contains the models weights as raw binary values. The XML file describes the structure of the model, i.e. the layers and how they are connected as well as the hyperparameters such as kernel size, and references the weights within the BIN file by their offset and length. An example layer that references a convolution kernel in the BIN file is shown in Figure 2.9.

ONNX The ONNX format [BLZ+19] has been proposed as an versatile exchange format of neural networks. Its specification is open and it is supported by a variety of companies. OpenVINO directly supports neural networks in ONNX format without prior conversion to its own IR format.

Converting the IR Hyodo is actively developing the tool *openvino2tf* [Hyo] to convert models in IR to several other formats, including TensorFlow, ONNX and others. This enables users of other frameworks to use the models as well.

3 Related Work

In this chapter we first explain different attack scenarios and attacker's objectives in the context of model recovery. We proceed by presenting a variety of existing attacks to recover internals of neural network models (such as structure or parameters) through various means. Finally, we highlight possible defenses proposed in the literature.
Different surveys especially tailored to attacks aiming at the recovery of neural network models exist [IGGK19; CDGK21]. Isakov et al.'s extensive survey paper [IGGK19] covers a broad range of attacks, from remotely conducted API attacks to invasive attacks against neural network models. Recently, Chabanne et al. [CDGK21] released an up-to-date survey of different side-channel attacks aiming to extract the architecture of neural network models, including a list of possible countermeasures. Recovering the model's structure already greatly helps an adversary since different attacks allow to obtain a functionally equivalent model when the structure is known.

3.1 Threat Model/Objectives

3.1.1 Threat Model

We assume the NCS 2 to be used in an IoT scenario where the NCS 2 is connected to some low-power host (e.g., a Raspberry Pi). The host runs OpenVINO to convert and optimize models for the NCS 2, program the NCS 2 with models, and run inference tasks on the NCS 2. The attacker has physical access to the IoT setup. Therefore, she can perform Man-in-the-Middle attacks against the USB connection[1] and measure power consumption and EM emanation of the NCS 2. From a software perspective, the attacker can trigger inference tasks but may not execute any additional code. The inference model processed by OpenVINO and loaded onto the NCS 2 is considered confidential intellectual property that is unknown to the attacker. The attacker aims to learn the model.

3.1.2 Attacker's Objectives

When attacking neural network models, different objectives are considered:

- Model extraction: extracting the architecture, i.e., the number and types of layers.

[1] e.g, using devices like the miniSniffer2 [bug]

- Recovery of parameters: additionally, extract all the parameters such as weights and biases.
- Input recovery: recovering inputs that have been used to initially train the model, or generating inputs that yield to a specific behaviour.

A more recent, thorough survey of different architecture extraction attacks (and countermeasures against them) is compiled by Chabanne et al. [CDGK21]. The authors focus on architecture extraction attacks since due to the fact that if an adversary knows the architecture, it is easier to also reconstruct the parameters. This is accomplishable through more sophisticated attacks or by other deep learning methods. Jagielski et al. [Jag+20] consider three different *adversarial goals* for model extraction:

1. **Functionally Equivalent Extraction**: According to Jagielski et al., this is the strongest assumption, given only input-output pairs. Given a model M, the goal is to construct a model $\hat{M}$ such that for all inputs x it holds that $\hat{M}(x) = M(x)$ where $M(x)$ is the output of model M for a given input x. It does not consider specific constructs such as dead-end neurons that do not contribute to the output.
2. **Fidelity Extraction**: For some *goal similarity function* $S(p_1, p_2)$, the goal is to construct a model $\hat{M}$ that maximizes $\Pr[S(\hat{M}(x), M(x))]$. The authors give the *label agreement* as an example for a similarity function. Fidelity extraction is significantly easier than functionally equivalent extraction according to Jagielski et al. Note that to reach this goal, the newly crafted model is required to also misclassify inputs that are incorrectly classified in the original model.
3. **Task Accuracy Extraction**: The goal to achieve in this case is to match (or exceed) the accuracy of the original model, i.e., maximizing $\Pr[\hat{M}(x) = y(x)]$ for all inputs x where $y(x)$ is the real classification for x. Due to the relaxation that in this case the model is allowed to exceed the accuracy of the original model, this is the easiest goal to achieve out of the three goals.

3.1.3 Comparison to Block Ciphers

Carlini, Jagielski, and Mironov [CJM20] discuss that *model extraction* can be seen as the cryptanalysis of a block-cipher. Instead of mapping plaintext to ciphertext using a keyed function E_k, a neural network is a parameterized function f_θ that maps inputs to outputs, i.e., labels. They compare adaptive chosen-plaintext attacks with model extraction. Since setting up a neural network usually does not involve any security considerations, they argue that it should be even easier to perform their attack. According to their paper, there are three main differences between classical cryptanalysis and model extraction.

1. The main objective differs: cryptanalysis may be considered as successful even without recovery of any key bits. This does not hold for model extraction where the attack is only successful if values are recovered.
2. The input and output size of neural networks do not match the usual symmetric encryption algorithms. The inputs are high-dimensional whereas the outputs are low dimensional. The authors argue that it might be more precise to compare model extraction with cryptanalysis of a MAC.
3. Neural networks usually process fixed- or floating-point numbers and operate using the corresponding arithmetic whereas symmetric encryption usually makes use of finite field integer arithmetic. This makes the attack of neural networks even easier since a bit-perfect recovery of values isn't necessary.

3.2 Attacks

Several works have already analyzed machine learning algorithms and the devices they run on with respect to attacks requiring physical access. We first classify these attacks by their approach. Then, for each attack class, we list and explain the related work, including their results.

Power Attacks (Xiang et al.; Wei et al.; Maji, Banerjee, and Chandrakasan)

We found three publications related to power analysis in the context of neural networks. Xiang et al. [Xia+19] discuss power side channel analysis to recover the internals of a Deep Neural Network. By observing the power consumption while inference is performed using different models, they obtain a *power feature set* consisting of mean, median and standard deviation with respect to the power consumption for each model. The power feature set is used to train a classifier. This classifier can then be used to identify which model has been used for future power observations.
More recently, Maji, Banerjee, and Chandrakasan [MBC21] recover the model's structure and parameters through SPA and DPA on different microcontrollers.
Additionally, Wei et al. [Wei+18] succeed to recover input images of a trained CNN by using power analysis against a FPGA-based hardware accelerator that executes the corresponding model.

EM Attacks (Batina et al.; Chmielewski and Weissbart; Yu et al.; Takatoi et al.)

Batina et al. [BBJP19] show that a combination of timing and EM side-channel attacks against hardware targets (Atmel Atmega328P and ARM Cortex-M3) to recover internals of

neural network models are feasible. They were able to reconstruct the architecture including the number of layers and their dimensions as well as several other hyperparameters (e.g., the activation functions) and even the weights of individual neurons, therefore succeed to fully reverse-engineer the target model. Using DPA, they target the multiplication $m = x \cdot w$ where x is a known input and w is the secret weight to recover. Exactly recovering the value is not required. Instead, they recover the weight up to a (previously chosen) precision, which in their case is 0.01. They claim that this is the first generic, passive side-channel attack to recover neural network models running on microcontrollers. This attack is one example for a functionally equivalent extraction.
Chmielewski and Weissbart [CW21] achieved similar results for architecture extraction. Instead of targeting general-purpose embedded devices, they focus on the Nvidia Jetson Nano as the hardware target that ships with a GPU and is a dedicated embedded ML accelerator.
Yu et al. [Yu+20] focus on binarized neural networks [Hub+16] which according to the authors are often used on edge/IoT devices. Binarized neural networks differ from regular neural networks in a way that both weights and outputs of the activation functions are constrained to -1 or +1. They first extract the network architecture by SEMA and utilize guessing heuristics based on timing SCA. Afterwards, they use different training sets to train a model with the given structure with a Generative Adversarial Network (GAN)-like approach. The resulting network achieves a very similar accuracy as the original network – as high as approximately 96% in their tests.
Takatoi et al. [TSSL20] suggest to use SEMA to identify the activation functions as an improvement over timing analysis as conducted in [BBJP19].

Timing Attacks (Duddu et al.)

Duddu et al. [DSRB18] show that timing attacks against neural network models are a serious threat. The authors assume a black-box scenario, i.e., no internal information of the model besides input and output dimensions are known. By feeding the measured execution times for different model architectures as training samples to different regressors (such as *Decision Trees* or *Support Vector Machines*), they obtain a classifier that is subsequently used to predict the layer depth of the model. Afterwards, several models with fixed layer depth but varying hyperparameters are trained using *Reinforcement Learning*. By limiting several hyperparameters to well-established values (e.g., by limiting the kernel size to either 3 or 5), the search space is drastically reduced. Then, the model with highest accuracy is being chosen. The accuracy of the resulting model is as close as 5% to the accuracy of the original model. They propose to stack their attack with cache attacks to further improve the accuracy.

Cache Attacks (Hong et al.; Yan, Fletcher, and Torrellas; Liu and Srivastava)

Besides timing, power and EM analysis, adversaries may have more direct access to the DUT. Cache attacks (applicable when the adversary can execute unprivileged code on the same target) have been shown to be efficient to recover the architecture of a DNN.
Hong et al. [Hon+18] presented a cache side-channel attack (*DeepRecon*) to recover neural network architectures, without actively querying the model. It is however required to run *DeepRecon* the attack on the system that the model is running on. *DeepRecon* monitors accesses to the deep learning library (e.g., TensorFlow) that the victim model is using through a cache side-channel technique called *Flush+Reload* and thereby is able to reconstruct the network's structure.
Yan, Fletcher, and Torrellas [YFT20] exploits the cache side-channel techniques *Prime+Probe* as well as *Flush+Reload* to recover model's structure using *CacheTelepathy*. Instead of tracing the library functions, they focus on GEMM (generalized matrix multiplications) that are a large component of neural networks.
Another attack called *GANRED* by Liu and Srivastava [LS20] employs a *GAN* in conjunction with cache attacks to recover the structure of a model.

HERMES (Zhu et al.)

The HERMES attack by Zhu et al. [ZCZL21] is a novel attack to recover the internals of a machine learning model, targeting the unencrypted PCIe communication between the host CPU and GPU. They claim that this is the first black-box attack to steal the whole model. This very powerful attack is able to recover the architecture *and* parameters of the model, but mainly applies to high-end platforms with a powerful GPU.
By reverse-engineering both the used library (CUDA) and the PCIe communication, they learn about the internals. The attack is split into an offline and an online phase. During the offline phase, the adversary learns which commands, kernels, addresses etc. belong to which layer type and builds a knowledge database out of it. During the online phase, the sniffed PCIe traffic is analyzed. By utilizing the database from the offline phase, the original network's architecture, its hyperparameters and its parameters are reconstructed. Both phases consist of a traffic processor for sorting out-of-order PCIe packets, an extraction module to filter out commands that are of interest and a reconstruction module that is first used to build up the knowledge database and later used for network reconstruction.

Membership Inference Attacks (Shokri et al.)

Another type of attack is the *membership inference attack*. In this scenario, an adversary's task is to determine whether a given input has been used to train a given model or not.

Shokri et al. present an attack [SSSS17] that works even in a remote black-box context, i.e., an adversary that only has access to the output of a model for a given input. They exploit the fact that a model behaves differently, depending on whether an input has been used as a training point or not. Based on the output distribution for a set of inputs, they train new *shadow models* on the same platform (e.g., same cloud provider) that aim to match the original model's probability distribution. The final attack model aggregates results from multiple *shadow models* and thereby learns to distinguish new inputs from inputs that have been used to train the original model.

3.3 Countermeasures

Due to the effectiveness of these attacks on machine learning, countermeasures have also been studied. Isakov et al. [IBCK18] point out timing side-channel countermeasures and propose power-efficient obfuscation methods. A non-exhaustive list of other countermeasures proposed in the literature include detection mechanisms added to the network [JSMA19], watermarking [Adi+18], homomorphic encryption [Dow+16] and garbled circuits [RRK18]. Dubey, Cammarota, and Aysu [DCA20] first describe a DPA attack against hardware inferencing. Afterwards, they propose mask-based countermeasures against first-order DPA attacks. Hua et al. [HUZS20] introduce a memory protection scheme called *MgX* for secure DNN hardware acceleration. Boemer et al. integrated homomorphic encryption into nGraph, calling it *nGraph-HE* [BLCW19] and *nGraph-HE2* [BCCW19] respectively. While several countermeasures have been discussed, most only address specific attacks and do not address the full range of threats that embedded machine learning engines face.

4 USB Sniffing Attack

All experiments were conducted with the most recent OpenVINO version which at the time of writing is *2021.4*. We use Ubuntu Linux 18.04 and Python 3.7 on our host computer since these are the officially supported versions by the most recent OpenVINO version. We first describe our USB sniffing setup in Section 4.1. Then, we observe how the model is transferred to the NCS 2 and show how to interpret this data in Section 4.2. Finally, in Section 4.3 we show how to reconstruct the original OpenVINO IR from the results we obtained before.

Threat Model The setup assumed for this section is shown in Figure 4.1. The attacker is in a Man-in-the-Middle position between the host and the NCS 2, capable of capturing the USB traffic between them. In addition, we allow the attacker the capability of triggering the setup process (e.g., by power cycling the host or disconnecting and connecting the NCS 2) of the NCS 2 and running inference tasks.

4.1 Attack Setup

Setup of the NCS 2 with OpenVINO

When connecting the stick to a USB port on the host, the NCS 2 identifies as `03e7:2485` (Movidius Ltd., Movidius MyriadX) and registers as a USB 2.0 device. In order to enable inference tasks on the NCS 2, OpenVINO programs the Myriad X with a specific firmware file[1]. The firmware is based on RTEMS [Pro]. After programming is done the stick disconnects, boots the firmware, and reconnects again via USB 3.0 as `03e7:f63b` (Intel Corporation, VSC Loopback Device). The stick is now ready for receiving models to run infer requests on.

Loading the IR to the NCS 2

When loading the model using the inference engine, the model is represented as a CNNNetwork instance that builds upon nGraph in the memory of the host. nGraph then performs several reordering, optimization and compilation steps – e.g., converting fully-connected

[1] filename `usb-ma2x8x.mvcmd`, approx. 2 MB

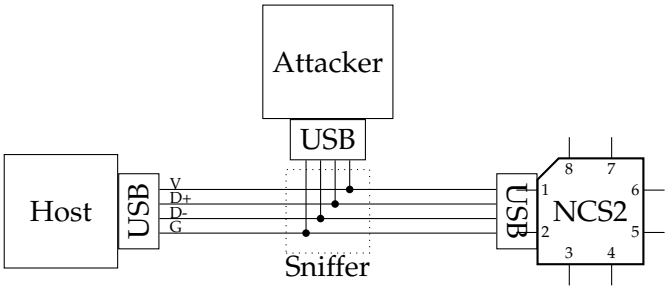


Figure 4.1: MitM attack: The attacker installs a USB sniffer between host and NCS 2 to capture the model while it gets programmed to the NCS 2.

layers to 2D-convolution layers, merging constant layers, or rearranging data, before converting the network into a binary blob that is understood by the Myriad X VPU. Afterwards, the binary blob is transferred to the NCS 2 in one burst transfer. The device opens two communication channels per network, one for the input and one for the output. Additionally, a watchdog thread periodically checks whether the stick still responds to ping messages. The device then listens for infer requests targeting that specific network. When an infer request arrives on the input channel, inferencing happens and results are sent back via the corresponding output channel. These channels are not exposed to the user explicitly but are rather used internally by OpenVINO. Due to the nature of those channels, the stick can manage multiple networks in parallel and since each of them can be addressed individually through their channels, the stick can perform inference on multiple networks simultaneously. When requesting OpenVINO to shut down, the device is rebooted, disconnects and reconnects in its USB 2.0 idle state again.

OpenVINO USB Communication Details

OpenVINO supports different log levels with `LOG_NONE` being the default log level. By setting the log level to `LOG_DEBUG`, during device initialization a lot of debugging info is displayed on the screen which helps to understand how the initialization process is working. OpenVINO is built on top of two other libraries for communication, namely `NCAPI` (low level NCS 2 API) and `XLink` (USB abstraction layer that builds upon *libusb* for communication with the device).

4.2 Setup and Operation of the NCS 2 with OpenVINO

The attacker uses a hardware USB sniffer to realize the MitM position as depicted in Figure 4.1 to capture the binary blob. An attacker with root privileges on the host can also use tools like Wireshark [Wir] to capture the USB communication. On a linux system,

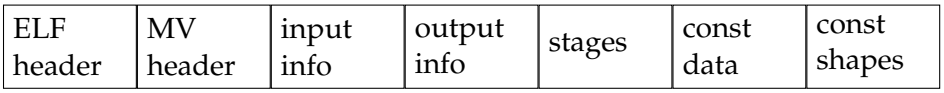


Figure 4.2: Binary data structure used to send models to the NCS 2.

the USB sniffing module utilizes the kernel module *usbmon*. However, we found that due to limitations of usbmon [boo], URBs (USB request blocks) larger than 2 MB cannot be captured. This fact poses a serious limitation as the binary blobs describing the CNN networks transferred to the NCS 2 easily grow beyond 2 MB in size. On a Windows host, these limitations can be circumvented by using another software (such as USBlyzer [USB]) to sniff the USB packets. An attacker with access to the OpenVINO API on the host can retrieve the binary file directly by calling the `export` function [Cora] of the executable network which is meant for debugging purposes.

4.2.1 Analyzing the Binary Data Blob Headers

The binary data that is sent to the stick when uploading a network is easily distinguishable from the status ping communication due to its larger packet size. We found the data to be organized in different segments and containing two headers: a modified ELF header that always contains the same data and a MV (movidius) header that partly depends on the network and the version of OpenVINO. An overview of the structure of the binary data is given in Figure 4.2. The MV header contains information about the number of stages as well as the input and output size of the network but not their shapes. Afterwards, an input and output section follows that describes the input and output layers of the network. These sections contain information about the corresponding input/output layer's name that is going to be used later for inference requests and responses. Additionally encoded is the location within the blob where the input/output shapes are stored. Then, a section describing the stages of the network follows. The last part is a constant data section that contains all the weights and biases which are referenced within the stage section. All the shapes that are referenced throughout the previous sections can be found just at the end of the constant data section. We still define it as a standalone section even though the starting point is not explicitly stated in the MV header.

4.2.2 ELF Header

The communication starts with a static header that looks like a modified ELF header. This header always contains the same values (for a fixed OpenVINO version). Table 4.1 shows all fields contained in the ELF header as well as their default values set by OpenVINO.

Table 4.1: ELF header.

Description	Offset	Type	Value
e_ident	0	bytearray	`0x7f`, 'e', 'l', 'f', 12 times `0x00`
e_type	16	uint16	1
e_machine	18	uint16	2
e_version	20	uint32	2
e_entry	24	uint32	0
e_phoff	28	uint32	0
e_shoff	32	uint32	0
e_flags	36	uint32	0
e_ehsize	40	uint16	416 (header size in *bits*)
e_phentsize	42	uint16	0
e_phnum	44	uint16	0
e_shentsize	46	uint16	0
e_shnum	48	uint16	0
e_shstrndx	50	uint16	0

4.2.3 Movidius Bin Header

The Movidius Bin Header follows directly after the ELF Header. This header is more interesting since it contains data that is actually dependent on the network, e.g. number of inputs. As can be observed in Table 4.2, it contains information about the input size and output size as well as offsets to four sections: the input section, the output section, the stage section and the constant data section. The Movidius Bin Header is padded with zeros until it is 64-byte aligned.

4.2.4 Input and Output Section

The input section precisely describes all the inputs of a model. Since OpenVINO currently only supports one input, this structure is sent exactly once. All fields except the `name` field have fixed lengths. The `name` field has a variable length; it is terminated by a zero-byte and is always padded to a multiple of 16 bytes using additional zero bytes. The output section looks exactly the same as the input section, except that it describes the outputs. Table 4.3

shows the section's fields.

4.2.5 Stage Section

The stage section (see Table 4.4) contains information about all the stages. Each layer of the original network is being converted to one or more stages. Stages are separated by the *Stage border symbol* which is defined as `0x7f83ff19` in the VPU plugin's source code. This bit pattern represents NaN when interpreted as a 32 bit IEEE 754 floating point number. Additionally, when interpreted as a 32 bit integer (both signed and unsigned), it is a prime number.

4.2.6 Data Section

The data section is implicitly split into two parts: constant data and constant shapes. Besides that, the section is not further structured but contains only the raw data. All data required by the model such as weights, biases etc. is written sequentially into the constant data part. All layer shapes are added to the end of the data section afterwards. Other sections reference the data by their offsets within this section. This section makes up by far the largest part of the transmitted data.

4.3 Reconstructing the IR

To the best of our knowledge, OpenVINO does not support the conversion from the binary format back to IR. This seems reasonable since not all transformations performed by OpenVINO before creating the binary blob can be reverted. Admittedly, it is possible to import the previously exported binary representation, but in this case the network can only be executed on the specific target the binary representation was compiled for, i.e. the NCS 2.
Due to the lack of official documentation for the data structures we gathered all information from the source code for the VPU plugin combined with additional reverse-engineering. For selected, smaller networks we manage to recover an IR of a model that is equivalent to the original model in IR modulo non-revertible optimizations. This enables us to run the extracted network on any other target supported by OpenVINO, i.e., CPU, GPU, or FPGA.
We begin by decoding the stage section to understand the shapes and connections of the hidden layers in the model. Each stage is described by a type, dependent parameters, and data. Most stages of interest come without any parameters or use default values. The data is partitioned into several *stage buffers*. The stage buffers that are present for all stage types are of great interest. Each stage has at least one input buffer and one output buffer

but additional buffers are possible. E.g. for convolution layers, three additional buffers are being used: for the kernel, the biases, and the additional scaling. The stage buffers precisely describe the location and representation of the data that is being processed by the corresponding stage. There are six different data locations: `none`, `input` (the infer request data), `output` (the infer response data), `blob` (the const data section of the binary blob), `bss` (buffer) and `cmx` (the on-chip memory that is shared between cores). In addition to the data location, the data type (`fp16` or `fp32` for 16-bit or 32-bit float respectively) as well as the shape and the data *offset* describing the exact location are given. Shapes are always located in the `blob` location, indicated as *constant shapes* in Figure 4.2. 32-bit floats are only being used in the Convert stages that downscale the input to 16-bit and upscale the output to 32-bit – all other stages in our experiments used the data type `fp16`.

We observe that most stages of interest such as convolution and pooling are implemented via the MyriadXHwOp stage type that is heavily parametrized. We therefore focus on understanding the MyriadXHwOp stage type. The first MyriadXHwOp parameter is the number of operations. Then, for each operation, an array of parameters follows, that states whether the stage performs a convolution or a pooling, the corresponding kernel size, padding strategy and size, stride, and whether a ReLU operation should follow. Since this stage is hardware-accelerated, it can distribute workload to multiple cores, which allows for offloading parts of the computation to different cores. Usually, each core performs the same operations but with different offsets, which makes sense for the convolution operation that moves the kernel over the input step by step. Since the computation of a layer depends on the outputs of the previous layers, operations crossing layer boundaries are performed sequentially.

At this point, we are able to fully reconstruct a model in IR from the binary data blob as we know the layers, their types, shapes, connections, weights, and biases. An overview of our reverse-engineering results is given in Table 4.5. But before evaluating the recovered model, we give an example for the reconstruction steps to allow the interested reader to reproduce our results more easily.

Example We give a short example for a subset of a very simple network that was trained on the MNIST data set. We only highlight relevant buffer information for the first three stages to give the reader an idea how data flow can be recovered. In Figure 4.3, the relevant parts of the original hexdump are highlighted. This allows the reader to follow our example more easily. Please note that the VPU uses little-endian representation for integers.

The network starts with a Convert (`0x6f`) stage (marked in red), using two buffers. We skip over the first bytes of the stage until the first highlighted value which indicates that the data type of buffer #0 is `fp32` (`0x03`). The next highlighted values tell us that the

```
00000140  78 00 00 00 6f 00 00 00  07 00 00 00 14 00 00 00  |x...o...........|
00000150  00 00 80 3f 00 00 00 00  00 00 00 00 01 00 00 00  |...?............|
00000160  03 00 00 00 43 00 00 00  02 00 00 00 03 00 00 00  |....C...........|
00000170  40 28 64 00 03 00 00 00  48 28 64 00 01 00 00 00  |@(d.....H(d.....|
00000180  00 00 00 00 40 0c 00 00  00 00 00 00 00 00 00 00  |....@...........|
00000190  43 00 00 00 02 00 00 00  03 00 00 00 40 28 64 00  |C...........@(d.|
000001a0  03 00 00 00 50 28 64 00  04 00 00 00 00 00 00 00  |....P(d.........|
000001b0  6f 00 00 00 19 ff 83 7f  60 00 00 00 13 00 00 00  |o.......`.......|
000001c0  00 00 00 00 04 00 00 00  00 00 00 00 21 43 00 00  |............!C..|
000001d0  04 00 00 00 03 00 00 00  78 28 64 00 03 00 00 00  |........x(d.....|
000001e0  88 28 64 00 04 00 00 00  00 00 00 00 00 00 00 00  |.(d.............|
000001f0  21 43 00 00 04 00 00 00  03 00 00 00 78 28 64 00  |!C..........x(d.|
00000200  03 00 00 00 98 28 64 00  05 00 00 00 00 f9 11 00  |.....(d.........|
00000210  13 00 00 00 19 ff 83 7f  2c 01 00 00 26 00 00 00  |........,...&...|
00000220  00 00 00 00 60 00 00 00  01 00 00 00 00 00 00 00  |....`...........|
00000230  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000240  04 00 00 00 01 00 00 00  02 00 00 00 03 00 00 00  |................|
00000250  00 00 00 00 20 00 00 00  05 00 00 00 05 00 00 00  |.... ...........|
00000260  01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000270  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000280  00 00 00 00 05 00 00 00  00 00 00 00 21 43 00 00  |............!C..|
00000290  04 00 00 00 03 00 00 00  78 28 64 00 03 00 00 00  |........x(d.....|
000002a0  98 28 64 00 05 00 00 00  00 f9 11 00 00 00 00 00  |.(d.............|
000002b0  21 43 00 00 04 00 00 00  03 00 00 00 a8 28 64 00  |!C..........(d.|
000002c0  03 00 00 00 b8 28 64 00  03 00 00 00 80 00 00 00  |.....(d.........|
000002d0  00 00 00 00 03 00 00 00  01 00 00 00 03 00 00 00  |................|
000002e0  c8 28 64 00 03 00 00 00  cc 28 64 00 03 00 00 00  |.(d......(d.....|
000002f0  00 00 00 00 00 00 00 00  03 00 00 00 01 00 00 00  |................|
00000300  03 00 00 00 c8 28 64 00  03 00 00 00 d0 28 64 00  |.....(d......(d.|
00000310  03 00 00 00 40 00 00 00  00 00 00 00 21 43 00 00  |....@.......!C..|
00000320  04 00 00 00 03 00 00 00  d4 28 64 00 03 00 00 00  |.........(d.....|
00000330  e4 28 64 00 05 00 00 00  00 19 11 00 26 00 00 00  |.(d.........&...|
```

Figure 4.3: Hexdump excerpt of the binary blob sent to the NCS 2 to configure an MNIST classifier. The dump shows the first three stages discussed in our example. The relevant content that is discussed in our example is highlighted.

data shape is stored in the `blob` (`0x03`) at offset `0x00642840` and indicate that the actual data is found at `input` (`0x01`) with offset `0x00`. The data type of the output buffer is `fp16` (represented by `0x00`) and the shape information is located at the same address as for the input, indicating that there is no change. The output location is `bss` (`0x04`) with offset `0x00`, indicating that the result should be stored in a temporary buffer. Each stage is terminated by the stage termination symbol `0x7f83ff19`. From this it is easy to reconstruct the first layer in our IR representation: It is of type Parameter (which is the layer type for network inputs) with the shape that we just learned.

The second stage (marked in green) is a Copy stage (`0x13`). This stage copies data of type `fp16` from `bss@0x0` (location of buffer #0) to `cmx@0x11f900` (location of buffer #1), so

the input information is stored in memory, accessible by all cores. We note however that the shape has changed to (1,1,28,28) through inspection of `blob@0x642878`. Therefore we reconstruct the corresponding stages that perform a reshape operation, namely a Const layer that loads information regarding the new shape as well as a Reshape layer.

The third stage (colored in blue) describes a MyriadXHwOp operation (`0x26`) that is configured to perform a convolution (`0x00`). We leave out the shapes to keep the example short. The stage parameters describe the convolution more precisely, e.g., that padding should be applied (`0x01`), the input comes via channel #0, output goes to channel #4, the kernel is loaded via channel #1, biases flow in via channel #2 and scales are loaded via channel #3. Moreover, the kernel size is 5x5, the stride is 1 and `withReLU` is set to `0x01` so that a ReLU operation follows after the convolution. Then, channel information follows as previously shown for the other stages. Input data comes from `cmx@0x11f900`, weights from `blob@0x80`, biases from `blob@0x0`, and scales from `blob@0x40`. Computed outputs are stored to `cmx@0x111900`.

A convolution is represented by multiple layers in the IR XML: A Const layer to load the convolution kernel, the actual Convolution layer, another Const layer to load the biases, an Add layer to apply the biases and, in most cases, a ReLU layer following afterwards. Since we learn the dimensions of the input, kernel, biases and output by observing the buffer information, the layers are also easily reconstructable. For the Const layers, we additionally reconstruct the weights by dumping the corresponding region from the blob into the reconstructed BIN file and set the right offset and size parameters within the XML file.

We keep track of the layers added to the XML file and insert the right connections between them. Therefore, by observing values being read from `blob`, we can reconstruct weights, shapes, etc. whereas observing values being read from and written to `cmx` allow us to follow the data flow throughout the network.

Further Notes

OpenVINO internally computes fully-connected layers using 2D-convolution. The same applies to 3D-convolution which is also implemented using several 2D-convolutions – those computations are equivalent [21]. In these cases, the original network has a MatMul layer whereas the reconstructed network has a Convolution layer. Therefore the reconstructed network is not identical to the original network, but provides equivalent accuracy.

We refer to [Hyo] for further converting the IR back to other frameworks such as TensorFlow, Caffe, ONNX and others, making the results obtained even more usable.

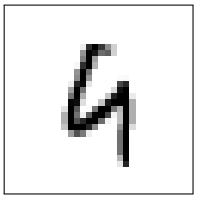


Figure 4.4: The only input that results in a different classification. It is labelled as '9' in the MNIST data set. The original model's prediction is '4' (49.95%), followed by '9' (48.41%). The reconstructed model's prediction is '9' (49.37%) followed by '4' (48.97%).

4.3.1 Evaluation of Reconstructed Machine Learning Models

We conduct our experiments with a simple MNIST classifier model. The input is a 28x28 pixel grayscale image of the handwritten digit. The output is a vector containing ten entries indicating the probability that the input image shows a certain digit.
We deploy the original model to the NCS 2 while observing the USB communication. We reconstruct the USB blob from the corresponding packets and feed the blob to our recovery script. The recovery script outputs both the model's structure and its weights.
We then iterate over the MNIST training and test set (60000+10000 images), supply them to the original model and record the outputs. The original model reaches an accuracy of 99.895% on the training set and 99.26% on the test set. We then deploy our reconstructed model to the NCS 2 and repeat the experiment. We observe that we reach the exact same accuracy of 99.895% on the training set and even a slightly increased 99.27% on the test set. Thus, with a single exception, the reconstructed model makes exactly the same predictions as the original model, i.e., they are (almost) functionally equivalent. The input that caused a difference is depicted in Figure 4.4. We assume that the different classification results are caused by rounding errors due to the low 16-bit floating point precision and the fact that the given input is ambiguous.

Table 4.2: Blob header.

Description	Offset	Type	Value
magic_number	52	uint32	`0x25ed`
file_size	56	uint32	file size in bytes (including all headers)
major_version	60	uint32	6
minor_version	64	uint32	0
num_inputs	68	uint32	1
num_outputs	72	uint32	1
num_stages	76	uint32	depends on the model
input_size	80	uint32	inference input bytes (32 bit float)
output_size	84	uint32	inference output bytes (32 bit float)
batch_size	88	uint32	depends on the model
bss_mem_size	92	uint32	
cmx_slices	96	uint32	depends on the model
shaves	100	uint32	depends on the model
has_hw_stage	104	uint32	boolean; depends on the model
has_shave_stage	108	uint32	boolean; depends on the model
has_dma_stage	112	uint32	boolean; depends on the model
input_section_offset	116	uint32	
output_section_offset	120	uint32	
stage_section_offset	124	uint32	
const_data_section_offset	128	uint32	

Table 4.3: Input/Output section.

Description	Type	Comment
input_index/output_index	uint32	
buffer_offset	uint32	
name_length	uint32	
name	bytearray	padded to a multiple of 16 bytes
shape_type	uint32	
shape_code	uint32	
shape_size	uint32	
shape_dims_location	uint32	
shape_dims_offset	uint32	
shape_strides_location	uint32	
shape_strides_offset	uint32	

Table 4.4: Stage section.

Description	Type	Value
stage_length	uint32	length of stage
stage_type	uint32	first time
num_shaves	uint32	number of shaves
params_pos	uint32	
stage_data	bytearray	depends on the stage
stage_type	uint32	second time
split_symbol	uint32	`0x7f83ff19`

Table 4.5: Summary of the reverse-engineering results for the binary data blob configuring an NCS 2.

Type	Value	Meaning
Buffer type	`0x00000000`	`none`
	`0x00000001`	`input`
	`0x00000002`	`output`
	`0x00000003`	`blob`: constant data section
	`0x00000004`	`bss`: temporal buffer
	`0x00000005`	`cmx`: shared buffer
Data type	`0x00000000`	`fp16`
	`0x00000003`	`fp32`
Stage type	`0x00000013`	Copy stage
	`0x00000026`	MyriadXHwOp stage
	`0x0000006f`	Convert stage
Separators	`0x7f83ff19`	Stage separator

5 Power Attack

In this chapter, we describe our experimental setup for power-based attacks in Section 5.1. Afterwards, we evaluate the hardware side-channel based attacks in Section 5.2.

5.1 Power Attack Setup

We use a Digilent Analog Discovery 2 oscilloscope (+ a Digilent BNC adapter) with a 14 bit resolution and a maximum sample rate of 100 MS/s and a maximum bandwidth of 30 MHz. We attach a modified USB 3.0 expansion cable with an exposed power wire with an additional resistor (1.1Ω) soldered in to measure the power consumption during operation of the DUT. The first probe is attached right after the resistor (when seen from host to DUT) and serves both as the power signal as well as the trigger for EM measurements. The power signal is a good trigger since the power consumption of the NCS 2 in its idle state is almost constant. A significant change can be observed during transmission of a model or while performing inference tasks. The second probe is a Langer MFA-K 0,1-30 [EMV] EM active probe that is placed onto the DUT. We amplify the EM probe's signal using a Mini-Circuits ZFL-1000LN amplifier. We removed the casing of the target to gain better access to the board.

An overview of our setup is given in Figure 5.1. Figure 5.5 shows how we positioned the EM probe.

5.1.1 Using the USB Connector as a Power Side-Channel

The power consumption of the NCS 2 can be measured by inserting a USB cable between the USB port of our main working device (i.e., IoT device or workstation) and the NCS 2 and solder in a resistor at the USB VBUS wire. The VBUS wire is one of the four wires that are present for all USB 2.0 ports. The other wires are D-, D+ and GND. Additional wires are available on a USB 3.0 cable for high-speed throughput, still the VBUS wire serves as the power supply. An undrilled USB extension cable is presented in Figure 5.2a. Figure 5.2b shows the reassembled USB 3.0 cable that contains an additional resistor at the VBUS wire.

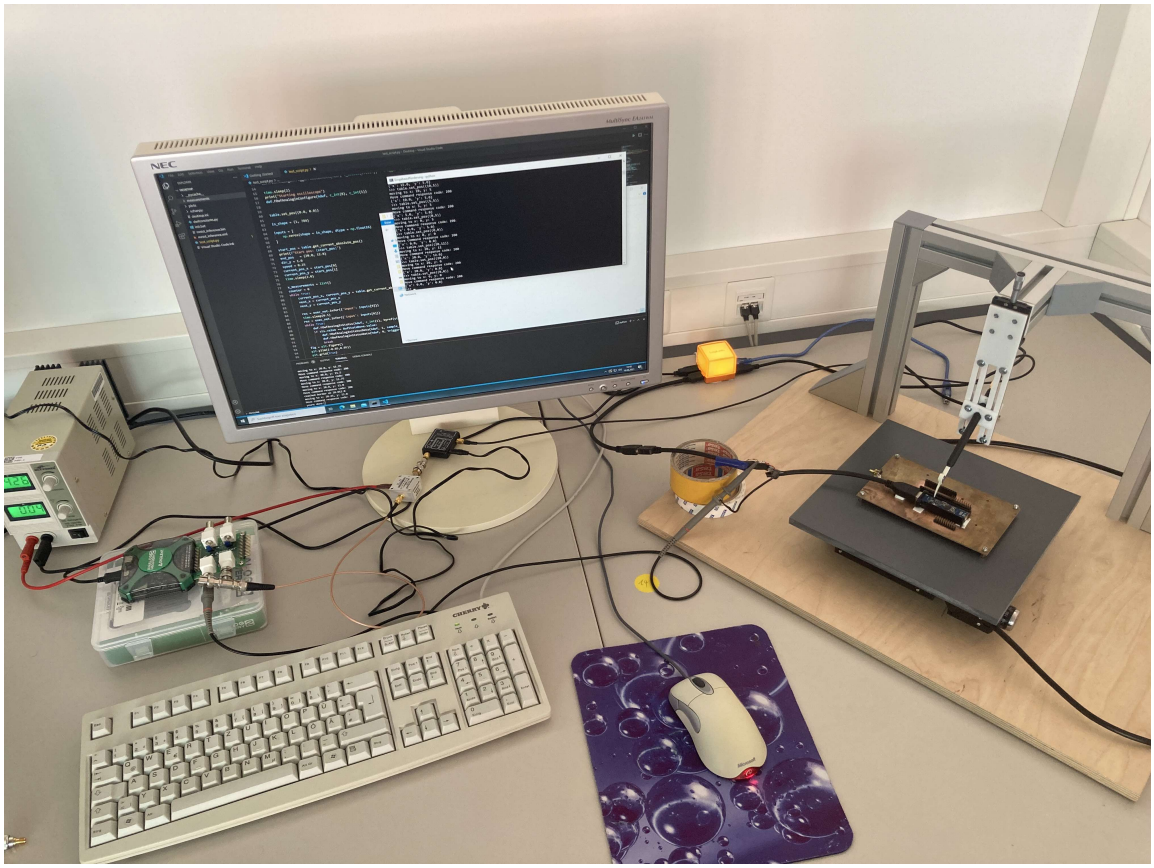
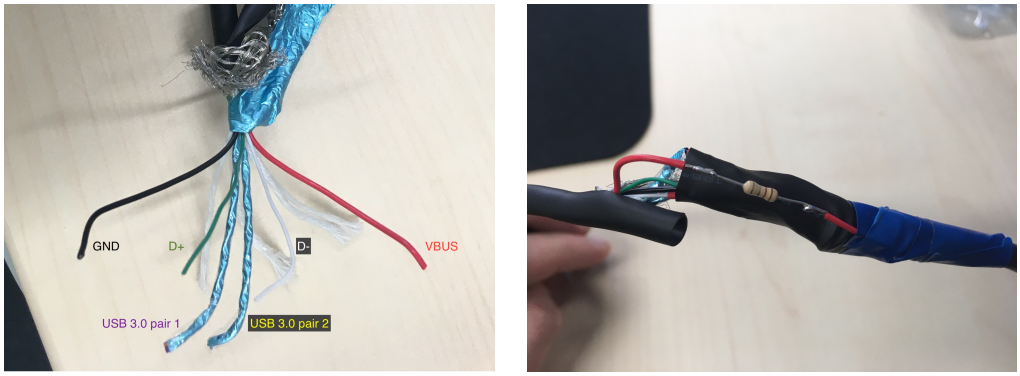


Figure 5.1: Our experimental setup. Depicted are: DC power supply, oscilloscope, signal amplifier, one EM probe, one power probe, the modified USB cable, the NCS 2, the measurement table.

5.1.2 Scanning the Target

To find leakage points on the NCS 2, we utilized the measurement table at the "Lab for secure hardware and software development" at the TH Lübeck [Lüb21]. The TH Lübeck granted us access to the measurement table which is controllable via a python library developed at TH Lübeck (`schanpy`) to enable precise, consistent movement of the DUT below the EM probe to find leakage points. We preload the NCS 2 with a simple MNIST classifier model and position the probe at the desired start point. Then the inference task is being executed, EM emanations are measured and the table is moved by a small offset. The process is repeated until the relevant chip area of the NCS 2 has entirely been scanned. We decided to use a stepping size of 250 µm, resulting in $52 \times 120 = 6240$ measurement points. After scanning, the results are being visualized via a heatmap shown in Figure 5.3.



(a) Undrilled USB 3.0 extension cable.

(b) Reassembled USB 3.0 extension cable with an additional 1.1 Ω resistor on the power VBUS wire.

Figure 5.2: USB 3.0 extension cable.

Interpretation

The heatmap in Figure 5.3 shows the Signal-to-Noise Ratio (SNR) for each measurement point where a lighter spot indicates a higher noise and a darker spot indicates lower noise. For each measurement spot, we performed one inference using a simple MNIST classifier and measure the power consumption as well as the EM emanation during inferencing. The power measurements serve as a trigger whereas the EM measurements are saved for further processing together with their physical board coordinates. First, we compute the mean value μ and the standard deviation σ of the trace's samples. Then, the SNR is computed as μ/σ.

The power chip (TPS 65266) is highlighted in the middle picture and the Movidus Myriad X chip is highlighted on the right picture by a green rectangle around each of them. Our experiments show that measurements taken near this chip are too noisy and should be avoided to obtain a clear signal. It is better to take a spot on the upper half of the stick which then results in a signal that is easier identifiable. Other metrics such as variance or the maximum sample value of each trace give similar results as the ones presented in Figure 5.3.

5.2 Power Attack Results

In the following section we examine whether hardware side-channel attacks such as timing attacks [Koc96] or simple power analysis [KJJ99] reveal secret information during the inferencing process on the NCS 2.

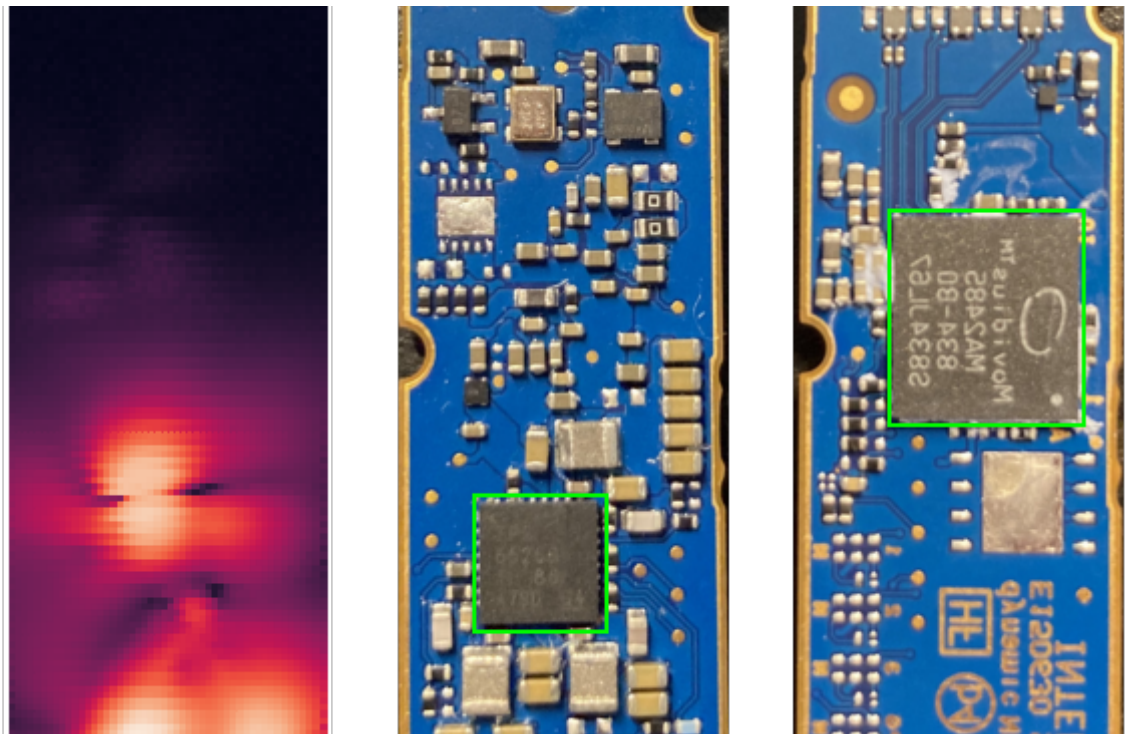


Figure 5.3: Left: SNR-heatmap of EM emanations. Middle: Top side containing the power chip (TPS 65266). Right: Bottom side containing the Movidius Myriad X chip. The image is flipped so that it matches the heatmap for better comparability.

5.2.1 Experimental Setup

The attack scenario for this section is shown in Figure 5.4. The attacker has physical access to the NCS 2 and is capable of measuring either the power consumption or EM emanations during operation.

5.2.2 Recovering the Model Structure

After queueing an inference request on the host, the OpenVINO library sends the input data to the stick and the device initiates the computation. We again use a simple MNIST classifier consisting of 2 convolutional layers for feature extraction, each followed by a pooling layer. Two dense layers perform the classification. The input consists of 784 floating point numbers describing the grayscale values of the input image, the output consists of 10 floating point numbers describing the probability that the input resembles the corresponding digit. By only sending parts of the model to the stick, a differing power consumption can be observed. Figure 5.6 shows several power traces as well as the corresponding EM traces that have been recorded by sending subsets of the model.

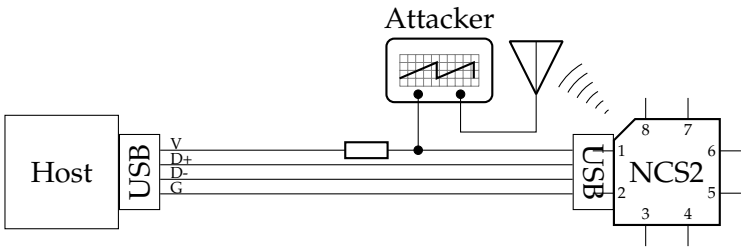


Figure 5.4: Power and EM attacks: The attacker measures voltage fluctuations in the supply voltage for and EM emanations from the NCS 2 with an oscilloscope.

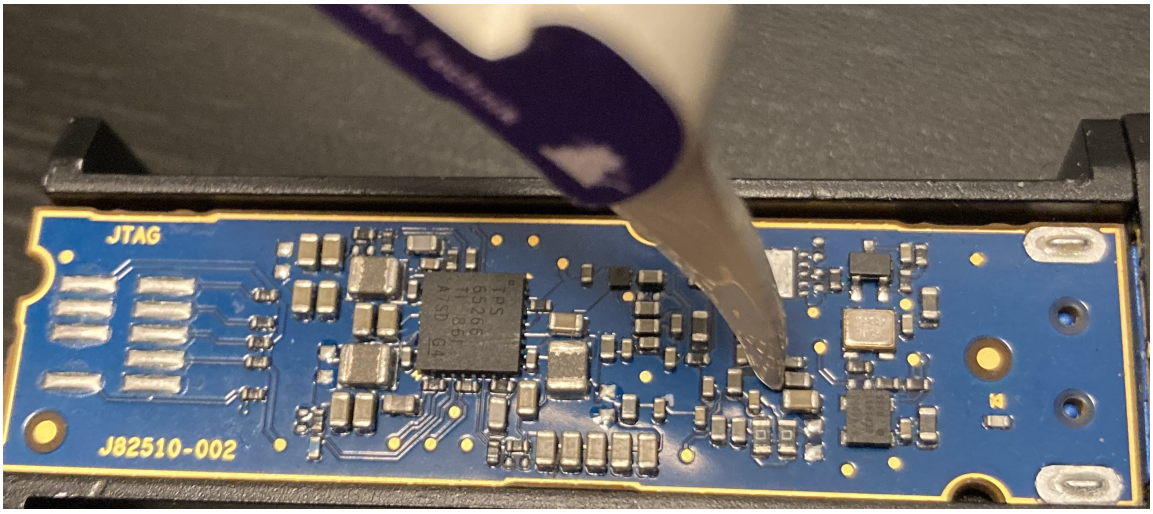


Figure 5.5: Positioning the Langer MFA-K 0,1-30 EM probe.

We use the peak (**3**) as a trigger. The upper trace has been recorded with a model that only computes the two convolutions, without pooling applied to the second convolution. The peak (**1**) occurs when computing the first convolution, peak (**2**) is caused by the first pooling layer and peak (**3**) occurs due to the second convolution. When the NCS 2 sends back the intermediate result, a pattern (**4**) is observed in the trace. When adding the second pooling layer to the network, a small peak (**5**) is observed that is similar to (**2**). Since peak (**1**) is smaller than peak (**3**), an adversary can conclude that the first convolution uses a smaller kernel than the second convolution. A transpose operation follows after the second pooling layer which is observable in the resulting power trace (**6**). In the next step, the first dense layer is added. This layer has huge impact on the power trace (**7**). Finally, the second dense layer (**8**) is added back to the network. Since the second dense layer is much smaller than the first one, it is hardly visible in the trace. Still, a difference is visible by comparing the position of the termination pattern (**5**) – which is shifted to the right and therefore not included in the figure anymore.

Our observations show that even a single power trace leaks critical details about the model's

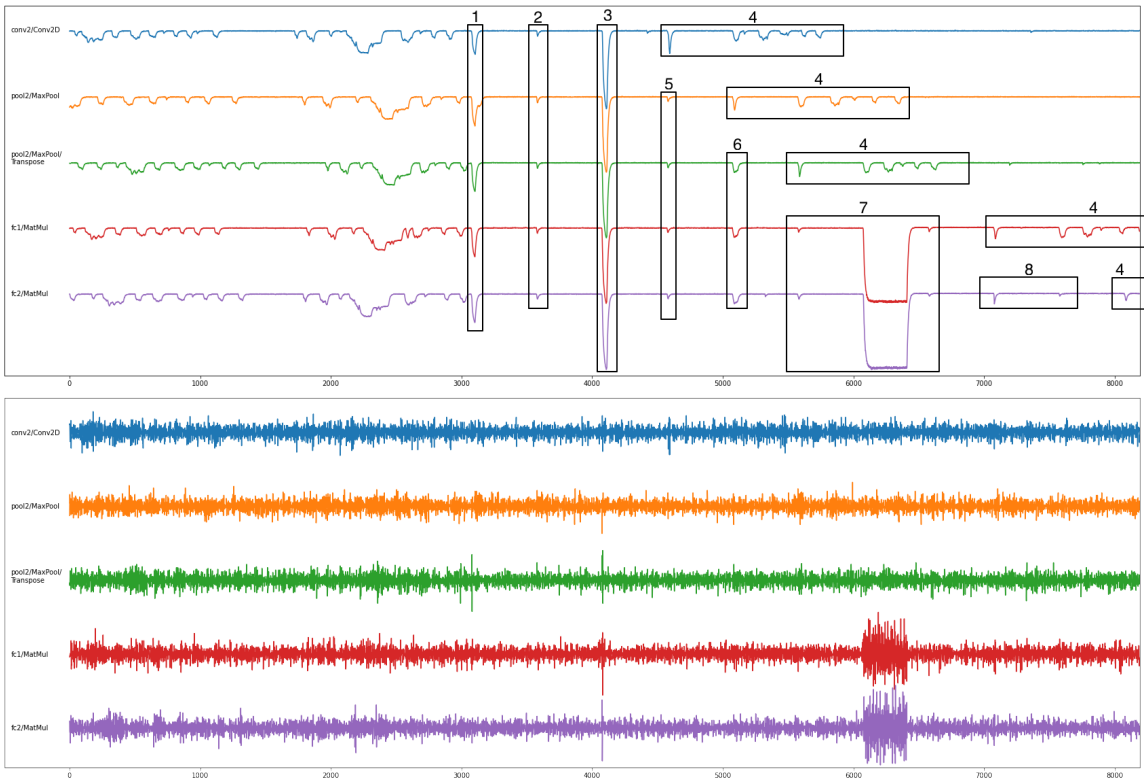


Figure 5.6: Comparison of power traces for different model stages. The upper plot shows power traces while the lower plot contains EM measurements of the same inference run.

structure. By observing the peaks, an adversary can recover the network's structure and, to some extent, estimate the layer's dimensions.

Due to the fact that fully-connected layers are expressed as convolutions as well, we cannot distinguish those types of layers. However this is not a problem since the outcomes of both layer types are equivalent.

6 Conclusions

6.1 Summary

In this thesis, we studied the vulnerability of the Intel NCS 2, a highly popular machine learning accelerator for embedded IoT workloads with respect to attacks requiring physical access. First, we familiarized ourselves with the device and investigated how the communication between it and the host is working. Then, we eavesdropped the USB communication. Subsequently, we created a heatmap of EM emanations which pinpoints spots of interest on the device. Lastly, we conducted further experiments to measure the power consumption during inference.
We recall the research questions that we specified in the introduction:

Research Question 1: *Does the NCS 2 leak secret information about the internal structure of the currently deployed model through side-channels?*
This research question can definitely be answered positively. We were successful in both the USB sniffing attack scenario as well as the SCA attack scenario. In the USB attack context, we were able to reconstruct the internal structure to an extent that allowed us to recover a network representation that achieves (almost) the same results as the original network. For this attack, not even a single inference request is needed. It is sufficient to observe the USB communication when the model is being transferred to the NCS 2.
In the SCA attack context, we were able to clearly distinguish layer boundaries by observing the power consumption and EM emanations of the DUT.
We note that the leakage is twofold: By capturing the USB traffic between the host and the NCS 2 while transferring the model allows an adversary to reconstruct the model through further reverse-engineering. By monitoring the power consumption or EM emanations of the NCS 2 during inference, different layers are distinguishable and the model structure can therefore be reconstructed by observing the patterns.
Even with a cheap oscilloscope, an attacker can learn the architecture of the model by eye-balling a single power or EM trace that was collected while the device performed an inference task.

Research Question 2: *Is it possible to reconstruct the entire model from side-channel leakages, if there are any?*

Unfortunately, the answer to our second research question is not as clear as it was for the first research question. As previously shown, it is possible to reconstruct the entire model by when observing the USB communication. Thus, for this kind of side-channel, the answer is positive. However, applying statistical methods such as DPA to recover a single weight from power or EM traces wasn't successful yet. The reason for this remains unclear due to time limitations.

6.2 Future Research Topics

We propose multiple future research directions here.

Parameter Recovery via DPA We propose to further investigate whether the power and EM leakages are exploitable to obtain further details regarding the model's parameters. To do so, we suggest to start with a more detailed leakage assessment (e.g., using the t-test) and then further analyze the corresponding leakage spots.

Multi-Model Deployment The NCS 2 is capable of running infer requests to multiple models simultaneously. We did not investigate whether our power attack still works in this scenario. We are confident however, that our USB attack still works in this case. Instead of targeting the infer requests, we target the model transfer directly. Since OpenVINO internally uses channels, it is still easy to distinguish multiple models. Therefore, reconstructing all models that have been transferred should be easily reconstructable.

Multi-Device Deployment It is also possible to stack multiple NCS 2 to share the workload between them. We only analyzed the single-device scenario and leave the multi-device scenario for future work.

Custom Layers OpenVINO supports *custom layers*. User can implement their own functionality by supplying OpenCL kernels suited to their needs. We did not investigate how a custom implementation of different operations might behave. Implementing operations via custom layers might defeat the approaches presented in this thesis in several ways:

- the power and timing characteristics may differ from the standard operations
- the custom layer could actively implement countermeasures by itself

Similar Devices Other devices (such as the Google Coral or the Nvidia Jetson Nano) might also leak confidential information about the deployed model. Thus, we propose to analyze physical attacks against them as well to establish better comparability of results.

Countermeasures The nature of the attacks we presented is inherently different which means that finding effective countermeasures is not trivial. Nevertheless, future work has to find solutions to protect IoT ML accelerators against physical and logical attackers alike while meeting power constraints and performance requirements. Countermeasures existing today may not meet these requirements as secret sharing or homomorphic encryption still come with huge overheads or performance losses.

A USB Blob Headers

```
0000   7f 65 6c 66 00 00 00 00 00 00 00 00 00 00 00 00   .elf............
0010   01 00 02 00 02 00 00 00 00 00 00 00 00 00 00 00   ................
0020   00 00 00 00 00 00 00 00 a0 01 00 00 00 00 00 00   ........ .......
0030   00 00 00 00 ed 25 00 00 80 4a 64 00 06 00 00 00   ....í%...Jd.....
0040   00 00 00 00 01 00 00 00 01 00 00 00 0e 00 00 00   ................
0050   40 0c 00 00 28 00 00 00 01 00 00 00 80 18 00 00   @...(...........
0060   09 00 00 00 07 00 00 00 01 00 00 00 01 00 00 00   ................
0070   01 00 00 00 c0 00 00 00 00 01 00 00 40 01 00 00   ....À.......@...
0080   80 1f 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0090   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00a0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00b0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00c0   00 00 00 00 00 00 00 00 10 00 00 00 69 6e 70 75   ............inpu
00d0   74 00 00 00 00 00 00 00 00 00 00 00 03 00 00 00   t...............
00e0   43 00 00 00 02 00 00 00 03 00 00 00 40 28 64 00   C...........@(d.
00f0   03 00 00 00 48 28 64 00 00 00 00 00 00 00 00 00   ....H(d.........
0100   00 00 00 00 00 00 00 00 10 00 00 00 6f 75 74 70   ............outp
0110   75 74 00 00 00 00 00 00 00 00 00 00 03 00 00 00   ut..............
0120   43 00 00 00 02 00 00 00 03 00 00 00 c4 2a 64 00   C...........Ä*d.
0130   03 00 00 00 dc 2a 64 00 00 00 00 00 00 00 00 00   ....Ü*d.........
0140   78 00 00 00 6f 00 00 00 07 00 00 00 14 00 00 00   x...o...........
0150   00 00 80 3f 00 00 00 00 00 00 00 00 01 00 00 00   ...?............
0160   03 00 00 00 43 00 00 00 02 00 00 00 03 00 00 00   ....C...........
0170   40 28 64 00 03 00 00 00 48 28 64 00 01 00 00 00   @(d.....H(d.....
0180   00 00 00 00 40 0c 00 00 00 00 00 00 00 00 00 00   ....@...........
0190   43 00 00 00 02 00 00 00 03 00 00 00 40 28 64 00   C...........@(d.
01a0   03 00 00 00 50 28 64 00 04 00 00 00 00 00 00 00   ....P(d.........
01b0   6f 00 00 00 19 ff 83 7f 60 00 00 00 13 00 00 00   o....ÿ..`.......
01c0   00 00 00 00 04 00 00 00 00 00 00 00 21 43 00 00   ............!C..
01d0   04 00 00 00 03 00 00 00 78 28 64 00 03 00 00 00   ........x(d.....
01e0   88 28 64 00 04 00 00 00 00 00 00 00 00 00 00 00   .(d.............
01f0   21 43 00 00 04 00 00 00 03 00 00 00 78 28 64 00   !C..........x(d.
```

Binary hex dump. Colors indicate: ELF header, MV header, Input section, Output section, Stage section (partially).

B Inference Engine Python Example Code

Before running the code, the OpenVINO environment needs to be set up properly. This is accomplished by running the `setupvars.sh` (or `setupvars.bat` when using Windows) shell script file that ships with OpenVINO.

```
from openvino.inference_engine import *
import tensorflow_datasets as tfds

# load the MNIST test set
mnist_test_ds = tfds.load('mnist', split='test')
inputs = list()
for img in mnist_test_ds:
    inputs.append(img['image'].numpy().astype('float32')/255.0)

# initialize the inference engine
core = IECore()
# load model into memory
net = core.read_network('network.xml', 'network.bin')
# initialize the target, and transfer the model
exec_net = core.load_network(device_name = 'MYRIAD', network = net)

# define inputs: use first three images from the MNIST test set
my_inputs = [img.reshape(1, 784) for img in inputs[:3]]
for img in my_inputs:
    # run the actual inference on the input
    result = exec_net.infer({'input': img})
    # print the results
    print(result['output'])
```

Outputs:

```
[[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

The outputs indicate that the first three test images represent the digits **2**, **0**, and **4**.

Glossary

AES Advanced Encryption Standard. xv, 16

ASIC Application-specific integrated circuit. 1

BNC radiofrequency connector, named after its designers (*Bayonet Neill Concelman*). 39

CNN Convolutional neural network. 10, 13, 23, 29

CPU Central Processing Unit. 19, 31

DPA Differential Power Analysis. xv, 15, 16, 23, 24, 26, 46

DUT device under test. 5, 15, 25, 39, 40, 45

ELF Executable and Linked File. xiii, 29, 30

EM Electromagnetic. xi, xv, 3, 15, 21, 23, 25, 39–46

FPGA Field-Programmable Gate Array. 19, 23, 31

GAN Generative Adversarial Network. 24, 25

GPU Graphics Processing Unit. 1, 19, 24, 25, 31

IE Inference Engine. 19

IoT Internet-of-Things. 2, 3, 21, 24, 39, 45, 47

IR Intermediate Representation. xi, xv, 19, 20, 27, 31, 32, 34

ML Machine learning. 1, 16, 18, 24, 47

MLP Multi-layer perceptron. xv, 6, 7, 10

MNIST Modified National Institute of Standards and Technology, a popular handwritten digits database used for machine learning. xv, 13, 14, 32, 35, 40–42

MO Model Optimizer. 19

MS Megasamples. 39

NCS 2 Neural Compute Stick 2. xii, xv, 2, 3, 5, 17–19, 21, 27–29, 31, 35, 38–43, 45, 46

ONNX Open Neural Network eXchange format. 19, 20, 34

OpenCL Open Computing Language, an open standard for parallel programming of heterogenous systems. 46

OpenVINO Open Visual Inference and Neural Optimization toolkit. xii, xv, 5, 18–21, 27–31, 34, 42, 46, 51

ReLU Rectified Linear Unit. 8, 9, 14

RTEMS Real-Time Executive for Multiprocessor Systems. 18, 27

SCA Side-Channel Analysis. 3, 16, 24, 45

SEMA Simple Electromagnetic Analysis. 15, 24

SHAVE Streaming Hybrid Architecture Vector Engine. 17

SNR Signal-to-Noise Ratio. xv, 41, 42

SPA Simple Power Analysis. 14, 15, 23

TPU Tensor Processing Unit. 1

USB Universal Serial Bus. xv, 2, 3, 17–19, 21, 27–29, 35, 39–41, 45, 46

VPU Vision Processing Unit. 2, 17, 31, 32

Bibliography

[21] *Visualize Model – OpenVINO Documentation*. 2021.4. Intel Corp. 2021. URL: https://docs.openvinotoolkit.org/2021.4/workbench_docs_Workbench_DG_Visualize_Model.html (cit. on p. 34).

[AAE] AAEON. *AI Solutions from AAEON powered by Intel Movidius Myriad X*. (Visited on July 5, 2021) (cit. on p. 17).

[Adi+18] Yossi Adi et al. "Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1615–1631. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/adi (cit. on p. 26).

[Ant+19] Mattia Antonini et al. "Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators". In: *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*. AIChallengeIoT'19. Association for Computing Machinery, 2019, pp. 49–55. ISBN: 978-1-450370-13-4. DOI: 10.1145/3363347.3363363 (cit. on p. 17).

[Ban] Imran Bangash. *NVIDIA Jetson Nano vs Google Coral vs Intel NCS: A Comparison*. (Visited on July 21, 2021) (cit. on p. 18).

[BBJP19] Lejla Batina et al. "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 515–532. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/batina (cit. on pp. 23, 24).

[BCCW19] Fabian Boemer et al. "NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 45–56. ISBN: 978-1-450368-29-2. DOI: 10.1145/3338469.3358944 (cit. on p. 26).

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–29. ISBN: 978-3-540-28632-5 (cit. on p. 15).

[BLCW19] Fabian Boemer et al. "NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. CF '19. Alghero, Italy: Association for Computing Machinery, 2019, pp. 3–13. ISBN: 978-1-450366-85-4. DOI: 10.1145/3310273.3323047 (cit. on p. 26).

[BLZ+19] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. 2019 (cit. on pp. 19, 20).

[boo] bootlin. *linux kernel 5.13 usbmon source code*. (Visited on July 5, 2021) (cit. on p. 29).

[bug] bugblat. *miniSniffer2 USB Protocol Analyzer - Hardware Sniffing on a Budget*. (Visited on July 5, 2021) (cit. on p. 21).

[CDGK21] Hervé Chabanne et al. "Side channel attacks for architecture extraction of neural networks". In: *CAAI Transactions on Intelligence Technology* 6.1 (2021), pp. 3–16. DOI: https://doi.org/10.1049/cit2.12026 (cit. on pp. 21, 22).

[Che+14] Sharan Chetlur et al. *cuDNN: Efficient Primitives for Deep Learning*. 2014. arXiv: 1410.0759 [cs.NE] (cit. on p. 1).

[Che+15] Tianqi Chen et al. *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. 2015. arXiv: 1512.01274 [cs.DC] (cit. on p. 19).

[Che+19] Mia Xu Chen et al. *Gmail Smart Compose: Real-Time Assisted Writing*. 2019. arXiv: 1906.00080 [cs.CL] (cit. on p. 1).

[CJM20] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. "Cryptanalytic Extraction of Neural Network Models". In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 189–218. ISBN: 978-3-030-56877-1. DOI: 10.1007/978-3-030-56877-1_7 (cit. on p. 22).

[Cora] Intel Corp. *ExecutableNetwork Class Reference – OpenVINO Documentation*. (Visited on July 5, 2021) (cit. on p. 29).

[Corb] Intel Corp. *Intel Movidius Myriad X Vision Processing Unit 4GB*. (Visited on July 5, 2021) (cit. on p. 17).

[Cor21] Intel Corp. *Using Encrypted Models with OpenVINO*. 2021. URL: https://docs.openvinotoolkit.org/2021.4/openvino_docs_IE_DG_protecting_model_guide.html (visited on July 20, 2021) (cit. on p. 19).

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, Çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 13–28. ISBN: 978-3-540-36400-9 (cit. on p. 16).

[CW21] Łukasz Chmielewski and Léo Weissbart. *On Reverse Engineering Neural Network Implementation on GPU*. Cryptology ePrint Archive, Report 2021/720. https://eprint.iacr.org/2021/720. 2021 (cit. on pp. 23, 24).

[Cyp+18] Scott Cyphers et al. *Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning*. 2018. arXiv: 1801.08058 [cs.DC] (cit. on p. 19).

[DCA20] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. "MaskedNet: The First Hardware Inference Engine Aiming Power Side-Channel Protection". In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 197–208. DOI: 10.1109/HOST45689.2020.9300276 (cit. on p. 26).

[Dow+16] Nathan Dowlin et al. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. Tech. rep. MSR-TR-2016-3. 2016. URL: https://www.microsoft.com/en-us/research/publication/cryptonets-applying-neural-networks-to-encrypte-data-with-high-throughput-and-accuracy/ (cit. on p. 26).

[DSRB18] Vasisht Duddu et al. "Stealing Neural Networks via Timing Side Channels". In: *CoRR* abs/1812.11720 (2018). arXiv: 1812.11720 (cit. on p. 24).

[EMV] Langer EMV-Technik. *MFA-K 0.1-30, Near-Field Micro Probe 1 MHz up to 1 GHz*. (Visited on July 5, 2021) (cit. on p. 39).

[Fou+06] Pierre-Alain Fouque et al. "Power Attack on Small RSA Public Exponent". In: *Cryptographic Hardware and Embedded Systems - CHES 2006*. Ed. by Louis Goubin and Mitsuru Matsui. Springer Berlin Heidelberg, 2006, pp. 339–353 (cit. on p. 15).

[Fuk80] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (1980), pp. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251 (cit. on p. 10).

[Glo] GlobeNewswire. *Movidius Supplies Myriad 2 Vision Processing Unit for Newest DJI Drone, the Mavic Pro*. (Visited on July 5, 2021) (cit. on p. 17).

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *Cryptographic Hardware and Embedded Systems — CHES 2001*. Ed. by Çetin K. Koç, David Naccache, and Christof Paar. Springer Berlin Heidelberg, 2001, pp. 251–261 (cit. on p. 14).

[Goo] Google. *Google Coral USB Accelerator*. (Visited on July 21, 2021) (cit. on p. 18).

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Springer Berlin Heidelberg, 2014, pp. 444–461. ISBN: 978-3-662-44371-2 (cit. on p. 14).

[Hon+18] Sanghyun Hong et al. "Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks". In: *CoRR* abs/1810.03487 (2018). arXiv: 1810.03487 (cit. on p. 25).

[Hub+16] Itay Hubara et al. "Binarized Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf (cit. on p. 24).

[HUZS20] Weizhe Hua et al. *MgX: Near-Zero Overhead Memory Protection with an Application to Secure DNN Acceleration*. 2020. arXiv: 2004.09679 [cs.CR] (cit. on p. 26).

[Hyo] Katsuya Hyodo. *openvino2tensorflow*. https://github.com/PINTO0309/openvino2tensorflow. (Visited on May 29, 2021) (cit. on pp. 20, 34).

[IBCK18] Mihailo Isakov et al. "Preventing Neural Network Model Exfiltration in Machine Learning Hardware Accelerators". In: IEEE, 2018, pp. 62–67. DOI: 10.1109/AsianHOST.2018.8607161 (cit. on p. 26).

[IGGK19] Mihailo Isakov et al. "Survey of Attacks and Defenses on Edge-Deployed Neural Networks". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–8. arXiv: 1911.11932 (cit. on p. 21).

[Inta] Intel. *Intel® Neural Compute Stick 2 Product Specifications*. https://ark.intel.com/content/www/us/en/ark/products/140109/intel-neural-compute-stick-2.html. (Visited on Sept. 25, 2020) (cit. on p. 2).

[Intb] Intel. *Neural Compute Stick 2 Data Sheet*. https://software.intel.com/content/dam/develop/public/us/en/documents/ncs2-data-sheet.pdf. (Visited on May 11, 2021) (cit. on p. 2).

[Intc] Intel. *Neural Compute Stick 2 Product Brief*. https://software.intel.com/content/dam/develop/public/us/en/documents/ncs2-product-brief.pdf. (Visited on May 11, 2021) (cit. on p. 2).

[Jag+20] Matthew Jagielski et al. "High Accuracy and High Fidelity Extraction of Neural Networks". In: *29th USENIX Security Symposium (USENIX Security 20)*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1345–1362. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/jagielski (cit. on p. 22).

[Jia+14] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014) (cit. on p. 19).

[Jou+17] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760 (cit. on p. 1).

[JSMA19] Mika Juuti et al. "PRADA: Protecting Against DNN Model Stealing Attacks". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019, pp. 512–527. DOI: 10.1109/EuroSP.2019.00044 (cit. on p. 26).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25 (cit. on pp. 3, 14, 15, 41).

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2 (cit. on pp. 3, 14, 41).

[Lam18] Paul Lambert. *SUBJECT: Write emails faster with Smart Compose in Gmail*. May 2018. (Visited on July 5, 2021) (cit. on p. 1).

[LBBH98] Y. LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324 (cit. on pp. 10, 13).

[LeC+89] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551 (cit. on p. 10).

[LS20] Yuntao Liu and Ankur Srivastava. "GANRED: GAN-Based Reverse Engineering of DNNs via Cache Side-Channel". In: CCSW'20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 41–52. ISBN: 9-781-450380-84-3. DOI: 10.1145/3411495.3421356. URL: https://doi.org/10.1145/3411495.3421356 (cit. on p. 25).

[Lüb21] Technische Hochschule Lübeck. *Lab for secure hardware and software development*. 2021. URL: https://www.th-luebeck.de/hochschule/fachbereich-elektrotechnik-und-informatik/labore/labor-fuer-sichere-hardware-und-software-entwicklung/uebersicht/ (visited on June 11, 2021) (cit. on p. 40).

[Lux] Luxonis. *DepthAI brings realtime Spatial AI to your product*. (Visited on July 5, 2021) (cit. on p. 17).

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/ (visited on May 23, 2021) (cit. on p. 19).

[MBC21] Saurav Maji, Utsav Banerjee, and Anantha P. Chandrakasan. "Leaky Nets: Recovering Embedded Neural Network Models and Inputs through Simple Power and Timing Side-Channels – Attacks and Defenses". In: *IEEE Internet of Things Journal* (2021), pp. 1–1. DOI: 10.1109/JIOT.2021.3061314 (cit. on p. 23).

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. 1st ed. Springer, Boston, MA, 2007, p. 338 (cit. on p. 15).

[MP43] Warren McCulloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147 (cit. on p. 5).

[NVI] NVIDIA. *NVIDIA Jetson Nano Developer Kit*. (Visited on July 21, 2021) (cit. on p. 18).

[Ope] OpenCV. *OpenCV AI Kit*. (Visited on July 5, 2021) (cit. on p. 17).

[Pov+11] Daniel Povey et al. "The Kaldi Speech Recognition Toolkit". In: *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Catalog No.: CFP11SRW-USB. Hilton Waikoloa Village, Big Island, Hawaii, US: IEEE Signal Processing Society, Dec. 2011 (cit. on p. 19).

[Pro] The RTEMS Project. *RTEMS Real Time Operating System (RTOS)*. (Visited on July 5, 2021) (cit. on pp. 18, 27).

[QS01] Jean-Jacques Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards". In: *Smart Card Programming and Security*. Ed. by Isabelle Attali and Thomas Jensen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 200–210. ISBN: 978-3-540-45418-2 (cit. on p. 3).

[Riv+18] Sergio Rivas-Gomez et al. "Exploring the Vision Processing Unit as Co-Processor for Inference". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 589–598. DOI: 10.1109/IPDPSW.2018.00098 (cit. on p. 17).

[RMN09] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. "Large-Scale Deep Unsupervised Learning Using Graphics Processors". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 873–880. ISBN: 9781605585161. DOI: 10.1145/1553374.1553486 (cit. on p. 1).

[RRK18] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. "Deepsecure: Scalable Provably-Secure Deep Learning". In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 978-1-450357-00-5. DOI: 10.1145/3195970.3196023. URL: https://doi.org/10.1145/3195970.3196023 (cit. on p. 26).

[Sch15] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (Jan. 2015), pp. 85–117. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2014.09.003. arXiv: 1404.7828 (cit. on p. 5).

[Sch19] Johan Schalkwyk. *An All-Neural On-Device Speech Recognizer*. Mar. 2019. (Visited on July 5, 2021) (cit. on p. 1).

[SSSS17] Reza Shokri et al. "Membership Inference Attacks Against Machine Learning Models". In: *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017, pp. 3–18. DOI: 10.1109/SP.2017.41 (cit. on pp. 25, 26).

[Tea17] Apple Inc. Computer Vision Machine Learning Team. *An On-device Deep Neural Network for Face Detection*. Nov. 2017. (Visited on July 5, 2021) (cit. on p. 1).

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23.1 (2010), pp. 37–71 (cit. on p. 14).

[Tra+16] Florian Tramèr et al. "Stealing Machine Learning Models via Prediction APIs". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 601–618. ISBN: 978-1-931971-32-4 (cit. on p. 1).

[TSSL20] Go Takatoi et al. "Simple Electromagnetic Analysis Against Activation Functions of Deep Neural Networks". In: *Applied Cryptography and Network Security Workshops*. Ed. by Jianying Zhou et al. Cham: Springer International Publishing, 2020, pp. 181–197. ISBN: 978-3-030-61638-0 (cit. on pp. 15, 23, 24).

[USB] USBlyzer. *USBlyzer - USB Protocol Analyzer and USB Traffic Sniffer*. (Visited on July 5, 2021) (cit. on p. 29).

[Wei+18] Lingxiao Wei et al. "I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. Association for Computing Machinery, 2018, pp. 393–406. ISBN: 9781450365697. DOI: 10.1145/3274694.3274696. URL: http://dx.doi.org/10.1145/3274694.3274696 (cit. on p. 23).

[WG18] Binghui Wang and Neil Zhenqiang Gong. "Stealing Hyperparameters in Machine Learning". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 36–52 (cit. on p. 1).

[Wir] Wireshark. *Wireshark*. https://www.wireshark.org. (Visited on May 23, 2021) (cit. on p. 28).

[Xia+19] Yun Xiang et al. *Open DNN Box by Power Side-Channel Attack*. 2019. arXiv: 1907.10406 [cs.CR] (cit. on p. 23).

[YF14] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 2014, pp. 719–732 (cit. on p. 14).

[YFT20] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2003–2020. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/yan (cit. on p. 25).

[Yu+20] Honggang Yu et al. "DeepEM: Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage". In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 209–218. DOI: 10.1109/HOST45689.2020.9300274 (cit. on pp. 23, 24).

[ZCZL21] Yuankun Zhu et al. "Hermes Attack: Steal DNN Models with Lossless Inference Accuracy". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhu (cit. on p. 25).

[Zha+20] Yuheng Zhang et al. "The Secret Revealer: Generative Model-Inversion Attacks Against Deep Neural Networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 253–261 (cit. on p. 1).