BarraCUDA: Edge GPUs do Leak DNN Weights

Peter Horvath
Radboud University

Lukasz Chmielewski
Masaryk University, Radboud University

Léo Weissbart
Radboud University

Lejla Batina
Radboud University

Yuval Yarom
Ruhr University Bochum

Abstract

Over the last decade, applications of neural networks have spread to every aspect of our lives. A large number of companies base their businesses on building products that use neural networks for tasks such as face recognition, machine translation, and self-driving cars. Much of the intellectual property underpinning these products is encoded in the exact parameters of the neural networks. Consequently, protecting these is of utmost priority to businesses. At the same time, many of these products need to operate under a strong threat model, in which the adversary has unfettered physical control of the product. In this work, we present BarraCUDA, a novel attack on general-purpose Graphics Processing Units (GPUs) that can extract parameters of neural networks running on the popular Nvidia Jetson devices. BarraCUDA relies on the observation that the convolution operation, used during inference, must be computed as a sequence of partial sums, each leaking one or a few parameters. Using correlation electromagnetic analysis with these partial sums, BarraCUDA can recover parameters of real-world convolutional neural networks.

1 Introduction

The field of machine learning has seen an explosive increase in interest and use over the last decade. In particular, deep learning has proven to be a versatile technique that provides state-of-the-art performance for many real-world applications. The use of Deep Neural Networks (DNNs) has proved useful for a broad range of domains, including playing chess [57], object detection [39], image classification [17, 26, 34, 38, 58], audio processing [49], forecasting [36, 52, 54, 55] and natural language processing [46]. Thus, deep learning applications have become indispensable and are changing many aspects of our everyday lives.

Deep learning typically employs artificial neural networks consisting of multiple layers of (simulated) neurons. When designing a deep learning solution for a problem, the designer first chooses the network architecture, which specifies the layers of neurons, including their sizes, types, and activation functions, as well as how the neurons are connected, i.e., which neurons' outputs are connected to which inputs. The designer then trains the network, selecting the weights used for each weighted sum and bias values that are added to the sums prior to the computation of a non-linear activation function, such as rectified linear unit (ReLU) [23].

Training a network for any non-trivial example is a resource-intensive process. There is a need to curate a specialized dataset of correctly labeled samples that can be used for the training. Moreover, the training process often requires days and even weeks of computation on specialized high-performance hardware, such as large quantities of graphical processing units (GPUs), and the whole process requires specialized expertise that is now in high demand. Thus, to protect owners' IP and to defend against potential attacks, trained models are often considered trade secrets, which should be protected from undue disclosure.

At the same time, there is a substantial market incentive for pushing machine learning to edge devices such as intelligent cameras, autonomous vehicles, and drones. Consequently, trained models are being deployed under a threat model that allows adversarial physical access to devices and exposes them to side-channel attacks. Indeed, side-channel attacks against neural network implementations on CPUs have been demonstrated using both electromagnetic side-channel analysis [13] and microarchitectural attacks [62], among others. Similarly, commercial deep-learning accelerators on FPGAs have also been shown to be vulnerable to parameter extraction via power-analysis [25]. However, GPUs are the dominant hardware in the world of deep learning due to their performance and the software ecosystem that they provide to implement neural networks. The CUDA parallel computing platform allows developers to quickly and efficiently deploy DNNs on modern GPUs, whose applications have spread to many areas, from data centers to edge computing. However, side-channel attacks on GPUs are challenging due to their complexity and inherent parallelism. So far, attacks on GPU implementations have only succeeded in recovering the

network architecture but not the parameters [16, 29, 41].

Therefore, this work focuses on the following research question:

Are proprietary implementations of neural networks on GPU vulnerable to parameter extraction using side-channel analysis?

Our Contribution

This work answers the question affirmatively. We perform side-channel attacks on multiple GPU architectures, such as the Nvidia Jetson Nano [9] and Nvidia Jetson Orin Nano [7], recovering the weights and biases of real-world networks.

Specifically, our attacks collect multiple traces with random inputs and then use correlation electromagnetic analysis (CEMA) [14] adjusted to recover the model's parameters from these traces. We further demonstrate the success of our attack on different neural network layers, such as convolutional and dense layers with varying batch sizes.

Our attack relies on the observation that convolution computation, the core operation in many neural network implementations, cannot be computed at once. Instead, convolution is computed as a sequence of partial sums, each depending on the previous sum and a small number of weights and inputs. Consequently, assuming the previous sum and the inputs are known, the adversary can guess the weights, use these to predict leakage, and correlate the prediction with the measured side-channel trace. However, to carry out the attack, we need to overcome several challenges.

Partial Sums Identification: Convolutional and dense layers in neural networks can be implemented in multiple ways; one of the most commonly used ways is matrix multiplications [15]. In a differential SCA attack, knowing how the target algorithm is implemented is crucial. Therefore, we first reverse-engineer CUDA binaries produced by TensorRT to determine both the parameter representation, the relevant partial sums, and the weights they depend on.

Attack Localization: A second challenge is that for an effective attack, the attacker needs to localize the attack, both spatially and temporally. To overcome this challenge, we adapt techniques from the domain of side-channel attacks on cryptographic implementations. We use a variant of Test Vector Leakage Assessment (TVLA) [56] both for finding the best physical location for placing the probe and for finding the time during the operation of the neural network in which a specific neuron is evaluated. Our leakage detection analysis proved effective for different GPU architectures.

GPU architecture: The GPU inherently introduces a high amount of noise due to its Single-Instruction-Multiple-Thread (SIMT) [10] architecture, where many parallel threads are executing simultaneously. Furthermore, due to the GPU hardware implementation, the scheduling of threads is not guaranteed to be fixed in time, which makes SCA attacks (e.g., CEMA) much harder than on other platforms, such as FPGAs and microcontrollers, where the execution is deterministic and sequential. Moreover, unlike attacks on cryptographic implementations, where the attacker typically aims to recover a relatively short sub-key, we need to extract a large number of parameters for DNN model extraction. A large number of traces and efficient signal processing are required to overcome these challenges. To that end, we develop a CUDA-based implementation of CEMA to execute the attack an order of magnitude faster for large datasets with millions of traces.

Parameter Extraction: We apply our techniques to convolutional and dense layers in neural networks while also investigating the impact of batch size on our attack. Our attacks against convolutional layers target a variation of the baseline EfficientNet [59] model. We deploy the model on the Jetson Nano in FP16 precision and the Jetson Orin Nano in INT8 precision for the parameters. We successfully and efficiently extract parameters from both GPUs with different data types using our CUDA-based analysis tool. Overall, the attack against the Jetson Nano requires 10 days for collecting traces and one to two days for trace alignment with an input batch size of 1. On the other hand, the Jetson Orin Nano requires only 1 day for trace collection and one day for trace alignment with an input batch size of 1. An experiment with a larger batch size was also conducted on the Orin Nano, and it took 5 days to collect traces and align them with an input batch size of 16. Once aligned, parameters can be extracted at a rate of one weight in six minutes for FP16 and 5 minutes for INT8 weights. The whole process is highly parallelizable. Thus, attacks on moderate-size models are well within the capabilities of well-resourced adversaries.

Organization. The rest of this paper is organized as follows: After providing the necessary background on side-channel attacks and deep learning (Section 2), we describe the overview of our attack and our experiment setup in Section 3. In Section 4 we describe our procedure for profiling the implementation, identifying relevant partial sums, and localizing their leakage. Finally, we present our parameter extraction attack in Section 5. In addition, Section 6 covers the limitations, possible extensions, countermeasures, and related work.

2 Background

Here, we introduce the concepts of side channels and related techniques used for the attacks, and we provide some background on Nvidia's CUDA programming model and GPU architecture.

2.1 Deep Neural Networks

Deep neural networks (DNN) are universal function approximators [28] that solve tasks by learning from data. First, a model learns from training data; then it can be deployed to make predictions on new, unseen data. Two main components

influence the trained model's performance on unseen data: the architecture and the parameters. The *architecture* refers to the structure of the model, the types, and order of transformations that the model applies on its inputs to arrive at some output. These transformations are also commonly called *layers*, and their output can be tweaked by changing their internal *parameters*. During the training process, these parameters are tweaked so that the final output yields correct results.

Convolutional layer. A convolutional layer consists of small *kernels* that extract different features from the layer's input e.g., an image. The dimensions of these kernels are usually much smaller than the input's dimensions to be able to extract fine-grained details. Each kernel extracts different features with its own parameters: the weights and bias. Each of them calculates the convolution between their parameters and a small part of the input by sliding every kernel on the input with a certain step size. In this work, we demonstrate the extraction of parameters, the weights, and the bias from convolutional layers.

Dense layers. A dense layer in a neural network consists of nodes where each node outputs the weighted sum of all the inputs. In this paper, we demonstrate the extraction of weights from dense layers.

2.2 Side-Channel Analysis

Side-Channel Analysis (SCA) exploits unintended physical leakages of electronic devices to extract secret information processed by them [32, 33]. Such leakage can occur through various channels, including power consumption, electromagnetic emanations (EM), timing, optical, or sound. It might lead to leakage of various types of secret information, e.g., on data and instructions processed. In academic settings, Side-Channel Analysis (SCA) was first introduced in the 90's, targeting cryptographic implementations running on then popular but constrained cryptographic devices such as smartcards [32, 33] and SCA poses ever since a constant threat to the security of various embedded systems. In that context, SCA aimed to recover the secret keys used in the cryptographic implementations. In this work, we exploit the EM side channel emanating from a GPU platform on which a neural network is running, but instead of targeting secret keys, we show how to recover neural network secret parameters: weights and biases.

2.2.1 Electromagnetic Emanation

The electromagnetic emanations from a computing device correlate with the code and data the device processes. This correlation has been used to break cryptographic implementations [35, 51], reverse-engineer neural networks [13, 16] and eavesdrop on display units [22, 27, 40].

In Correlation EM Analysis (CEMA) [14], the attacker uses the correlation coefficient as a side-channel distinguisher, i.e., the statistical method used for the key recovery. Essentially, CEMA allows an attacker to recover parts of a secret that is used in a targeted operation by using a known plaintext attack: measured samples are correlated against a synthetic leakage value (i.e., leakage model) that is generated from an intermediate value calculated for all possible values of a (part of) the secret. In our case, all CNN parameters (i.e., weights and bias) are considered as the whole secret, and a single weight or bias is considered to be a single target of CEMA that needs to be repeated for all of them; the intermediate values are results of computations within neurons. Observe that for the above approach to work, we need to assume that the inputs used in the computations are different and are known to the attacker.

Two commonly used leakage models in SCA are the Hamming weight (HW) model, which predicts that the leakage is linear with the number of set bits in the data (i.e. its Hamming weight), and the Hamming distance (HD) model, which predicts that the leakage is linear with the number of bits that flip between consecutive data values. HW leakage usually occurs in practice when a value is transferred via the system bus, and HD leakage occurs when an intermediate value stored in a register is overwritten with another value. We consider both of these models in this paper.

2.2.2 Leakage Assessment

For leakage assessment, which is a critical part of a security evaluation of a chip, we rely on the default techniques used in side-channel analysis, i.e., intermediate-value correlation and Test Vector Leakage Assessment (TVLA) [56]. The idea behind intermediate-value correlation is to consider the correlation traces generated for all possible values of the secret. There should be a correlation peak in the trace when the correct guess (for the secret) is processed because the guess is then in agreement with the leakage predicted. This is a consequence of the fact that processing different data causes different physical information, such as timing, power, or EM, often referred to as leakage. While this approach is often used in actual side-channel attacks, the disadvantage is that it requires a large number of traces, similar to the CEMA attack. TVLA or other leakage detection methods, like χ^2-test [44], are often more efficient (faster) as we control the parameters in the leakage detection phase. The main idea of TVLA is to check whether two distributions that process different intermediate values are equivalent or not using Welch's t-test. If the two groups are deemed equivalent, then the TVLA will not observe the leakage. Concretely, we verify whether two sets of measurements show significant differences if one set has a fixed weight and the other has random weights. We refer to this setting as "fixed versus random". Since TVLA searches for any leakage not necessarily exploitable by an

	Jetson Nano	Jetson Orin Nano
GPU architecture	Maxwell	Ampere
# SMs	1	8
# CUDA cores	128	1024
# Tensor cores	-	32
Max. clock	920 MHz	625 MHz
FP16	✓	✓
INT8	-	✓

Table 1: Comparison between the GPUs of Jetson Nano and Jetson Orin Nano.

attack like CEMA, it also usually requires much fewer traces than intermediate-value correlation. Therefore, for the sake of efficiency, we mainly use TVLA in this paper, especially for determining which part of the traces processes each weight using TVLA. We use intermediate-value correlation only to determine the best location of our probe[1] and for preliminary characterization of leakage patterns.

2.3 CUDA Programming Model

To leverage the parallelism offered by GPUs, Nvidia exposes the CUDA programming model [5] to developers. In this model, multiple abstraction levels exist and each level has different implications forGPU hardware. The lowest level of abstraction is the `thread`, which executes a CUDA function defined by the developer. The number of threads executing the CUDA function in parallel is specified at the time of invoking the function.[2] Subsequently, multiple threads can be grouped together into a single `block` of threads. Threads in a block have a per-block on-chip shared memory region where they can exchange data with other threads. Blocks of threads form a `grid` of thread blocks. Each block in a grid executes independently from other blocks, but all blocks in the grid share the same off-chip global memory region.

2.4 GPU Streaming Multiprocessor

When a CUDA function is invoked, the parallel threads execute on the GPU's Streaming Multiprocessor (SM). A GPU can consist of one or more SMs to improve parallelism further. In this paper, we mount our attack on two different GPU architectures, the Maxwell [11] and Ampere [7] GPUs embedded in a System-on-Chip (SoC). For both architectures, when blocks of threads are scheduled onto a particular SM, the threads in the blocks are divided into groups of 32 threads, also called `warps`. Every warp is assigned to a particular Processing Unit (PU) in the SM, and the warp scheduler in a PU is responsible for scheduling and issuing instructions for warps that are ready for execution at every clock cycle. Additionally, both architectures have 4 PUs per SM, each with dedicated resources (e.g., register file) to manage warps. This means that 4 warps can be issued instructions in parallel at every clock cycle in an SM. The Jetson Nano and Jetson Orin Nano's GPUs are similar, but significant differences are summarized in Table 1. Overall, the Ampere GPU is a larger and more capable GPU with more hardware support for different operations (matrix-multiply) and data types (e.g. INT8) that are heavily used in deep-learning inference.

[1] We simply check at which location the absolute t-test peak is the highest as in [20].

[2] Functions in CUDA are also called *kernels*. We use the term function to avoid confusion with kernels in CNNs.

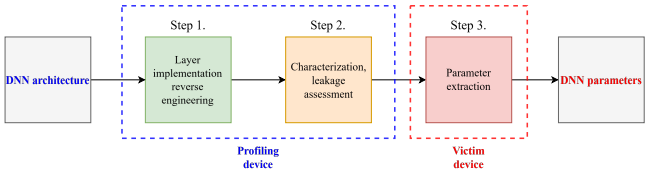


Figure 1: High-level description of the attack procedure.

2.5 TensorRT Workflow

TensorRT is a framework dedicated to accelerating neural network inference on GPU, using implementations from different libraries (CuDNN, CuBLAS, TensorRT) that are timed against each other to choose the fastest. We use the TensorRT framework in our experiments and demonstrate the parameter extraction attack on the implementations provided by the framework.

3 Attack Procedure

Our attack, depicted in Figure 1, assumes that the adversary knows or can recover [16, 29, 41], the DNN model architecture. The adversary aims to use side-channel observations to recover the unknown weights and biases used in the victim device. The attack follows two main phases. In the first, the adversary uses their prior knowledge to instantiate an equivalent DNN architecture on a profiling device with identical hardware, albeit without the weights. This profiling device is used to learn how information about weights and biases leaks. This knowledge is then used to guide the second phase of the attack, which recovers the desired information.

The attack builds on the observation that convolution is the core operation performed during inference. Specifically, A single $p \times q$ kernel with weights w, bias b and input feature map x calculates as follows:

$$c_{sum} = \mathbf{x} * \mathbf{w} = \sum_{i=1}^{p \cdot q} w_i \cdot x_i = w_1 \cdot x_1 + \cdots + w_{p \cdot q} \cdot x_{p \cdot q}, \quad (1)$$

$$c_{out} = c_{sum} + b, \quad (2)$$

where c_{sum} is the result of the convolution between the weights and the input, while c_{out} is the output of the kernel

(if the kernel contains a bias). Finally, if the layer includes an activation function f then it is applied after the convolution:

$$c_f = f(c_{out}).$$

Since the convolution in Equation 1 cannot be computed in a single step, its computation consists of a series of intermediate computations, each depending on one on a small number of weights. Our attack targets these intermediate computations, looking for correlations between the results of the computations and the side-channel observations. We refer to these results as *intermediate values*.

The profiling phase aims to *identify* the intermediate values used in the implementation and to *localize* at which time points they leak in the traces. The attack then performs correlation analysis on the side-channel information at the identified locations to recover the weights.

The rest of this section outlines our threat model and describes the experimental setup. The attack itself is described in two sections. First, Section 4 describes the profiling phase. Then, Section 5 describes and evaluates the recovery of weights and biases from the victim device.

3.1 Threat Model

Our attack targets edge devices that execute machine learning inference for their functionality. The attacker aims to recover the trade secrets encoded in the parameters (weights and biases) of the machine learning model, for example, in order to steal the IP that encodes them or as a step in designing an adversarial attack on the machine learning model.

We assume that the target device operates correctly and that the code is secure so the attacker cannot exploit programming vulnerabilities to acquire the parameters. However, we assume that the attacker knows the architecture of the model, including the number of layers, their sizes and types, and how they interconnect. Attackers that do not have the information can use techniques developed in past works to recover the architecture [16, 29, 41].

As is typical for edge devices, we assume the attacker has unfettered physical access. In particular, we assume the attacker can open the device and place electromagnetic probes at locations that leak information. As we discuss in Section 3.3, our target devices have multiple leaky locations, allowing the attacker a choice.

Last, we assume that the attacker can monitor the electromagnetic emanations from the target device during the time that the device performs inference. The attacker needs to be able to observe the emanations over multiple sets of inputs and should also be able to choose these inputs if the target parameters have INT8 data to reduce complexity. Otherwise, FP16 parameters do not require control of the inputs.

3.2 Sensitive Intermediate Values

A successful CEMA attack against a target algorithm requires the attacker to find sensitive intermediate values that depend on secret data. In convolutional and dense layers, a sensible choice for these intermediate values is the *partial sums* that depend on the secret weights. The partial sums allow an adversary to attack one or a few weights at a time and, therefore, to reduce complexity. If an attacker targeted the final results of these layers, the complexity would increase as these results depend on many secret weights, making the attack infeasible.

However, the actual *implementation* of these layers, and subsequently the computation of partial sums, are dependent on the target layer and hardware characteristics. Some of the important layer characteristics are the number of input and output channels in a layer and the used representation for the parameters. In addition, hardware characteristics such as the number of SMs and shared and global memory size can also influence this implementation. Consequently, different layer and hardware configurations can lead to slightly different implementations. Although these implementations can have identical structures, fine-grained details such as partial sum computations can differ. Therefore, an attacker first needs to reverse engineer how each layer of the target DNN architecture is implemented on the target device. This task can be aided via reverse-engineering of the GPU framework and analysis of leakage of the profiling device.[3]

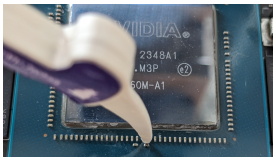
3.3 Experimental Setup

To gather side-channel information, we collect EM traces as they are less invasive and can provide more localized information than power measurements. It is also closer to the real world as fewer modifications to the chip are required. Our two targets, the Jetson Nano and Jetson Orin Nano, are similar in that both feature a SoC mounted onto a PCB in a flip-chip package. However, the Jetson Orin package is significantly larger than the Jetson Nano's while also surrounded by more capacitors. In order to access the packages, we remove the heatsinks from both devices and use an external fan to provide cooling to the devices. In our setups, the GPU cores operate at the highest supported clock frequencies[4] as shown in Table 1. We use the Lecroy 8404M-MS oscilloscope at a sampling rate of 10GS/s with a Langer MFA-R 0.2-75 near-field probe [1] to collect electromagnetic traces. In our experiments, we find that we need a sampling rate of at least 5 GS/s to see leakage. Increasing the frequency up to 20 GS/s does not improve the results. In the FP16 case, the number of samples in a single trace for the first convolutional layer is 400 000 with a batch size of 1. In the INT8 case, it is 150K per trace.

[3] One can also imagine an attacker without a profiling device searching through all the possible intermediate values to find the correct one. This approach is feasible but time-consuming, and we do not follow it.

[4] At the time of submission, the maximum supported clock frequency was 625 MHz for the Jetson Orin Nano.

(a) Location of Langer EM probe between two capacitors on the Jetson Nano.



(b) Location of Langer EM probe between 3 capacitors on Jetson Orin Nano.

Figure 2: EM probe locations for the Jetson Nano and Jetson Orin Nano devices for successful parameter extraction attacks.

3.3.1 EM Probe Positioning

To find the best position for the EM probe, we use both TVLA and intermediate-value correlation experiments. Specifically, we instantiate models whose architectures are identical to the target model for each possible location. We then capture two sets of traces. The weights and inputs are always the same in the "fixed" set of traces. In the "random" set of inputs, we select a random value for one of the weights in the model, and the inputs are the same. We then use TVLA to measure the statistical difference, expressed as the t-value, between the set of traces and intermediate-value correlation to find the correlation between the random weights and the leakage. Significant correlation and t-value peaks indicate a strong signal.

Jetson Nano We observe several promising locations for placing the EM probe. Figure 2a shows one such location between two capacitors in the power-supply circuit of the Jetson Nano. Additionally, using XY scan, we find several locations that exhibit similar leakage on the surface of the Jetson Nano's SoC. Parameter extraction is possible from both the best location on the SoC and between the capacitors. In the rest of this paper, we use the capacitors' location because probe placement is easier and does not require a detailed scan.

Jetson Orin Nano On the Jetson Orin's chip surface, we cannot pick any GPU-related signal. However, some nearby capacitors still leak information related to GPU activity. Figure 2b shows one vulnerable spot that allows for parameter extraction. With further experimenting over the chip surface, we find that probes sensitive to higher frequencies, such as the Langer RF-B 0.3-3 [2], can pick up exploitable signals over the chip as well. In this paper, the presented results use the location shown in Figure 2b with the Langer MFA-R 0.2-75 probe.

3.3.2 Trace Acquisition

In order to extract the parts of the traces related only to the inference operation, we used `nsys` from the CUDA Toolkit to get information about the execution times of the operations on the GPU. Therefore, we used the Lecroy oscilloscope's SmartTrigger feature to trigger on the rising edge of the first layer as it proved to be more reliable.

3.3.3 Trace Preprocessing

In general, the collected traces contain a lot of jitter and the clock of the cores is not stable, which can be confirmed by looking at the traces in the frequency domain. This makes the detection of time where the implementations leak and the subsequent CEMA attack harder. Since aligning the traces accurately with static alignment [42][5] at many locations at the same time is not possible, we use elastic alignment [60][6] to improve the leakage detection process. Although elastic alignment performs alignment at every time point, it comes at a price: it requires finding the optimal input parameters and is computationally expensive. Moreover, despite tuning the parameters, it decreases the amount of leakage in the traces. Therefore, it is used only to detect leaking points, but the CEMA attack is carried out on the raw traces after static alignment is applied on the leaky part of the trace.

4 Profiling

We now turn our attention to the profiling phase of the attack, in which the adversary uses a profiling device to analyze the model and identify trace positions in which partial sums leak. Profiling consists of two main steps. In the first step, the adversary analyzes the software that the GPU executes to identify partial sums that depend on weights and characterize the dependencies. Once these are found, in the second step, the adversary collects side-channel traces from the profiling device and uses statistical tools to find the leaking time points in the traces at which leakage of each weight and bias can be observed.

4.1 Layer Implementation

In this section, we discuss the high-level structure of the code that performs the operations of convolutional and dense layers in DNN models. As the code is unavailable to us, we use `cuobjdump` [6] to produce assembly code, which we can analyze. While each implementation is different, all convolutional implementations follow the same structure:

- (1) block of initialization instructions,
- (2) block of convolution operations, and
- (3) block of bias addition and ReLU calculation.

We now explain these three blocks in more detail.

(1) Init Block. The first main block consists of instructions that set up the CUDA function. This block initializes the 64

[5] Static alignment employs a standard pattern-based approach: we select a part of a trace as a reference and compute correlation for each offset within a chosen range for each of the traces. We then shift each trace by the respective offset that maximizes the correlation.

[6] Elastic alignment is a parametrized machine-learning-based technique to align the traces on all the distinctive patterns at the same time. However, this method tends to be error-prone and results in a decreased leakage in practice.

accumulator registers that are later used to store the convolution results (R0–R63). Higher registers (R64 and above) are used to load the weights and inputs. Since the GPU registers are 32-bit wide, each higher register is loaded with either two FP16 values or four signed 8-bit integers, depending on the data type used for the computations.

② **Convolutional Block.** The second block performs the convolution of the weights and the inputs, i.e., the partial sums are computed in this block. It consists of repeated vectorized loads and arithmetic instructions. A set of higher registers is assigned to be loaded with inputs and weights from different input channels. In addition, this block is executed multiple times depending on the hyperparameters of the convolutional layer, such as the kernel size.

③ **ReLU Block.** The third block adds the biases and performs the activations. For the FP16 implementation, the two partial sums of the accumulator need to be combined before adding the bias and calculating the ReLU. Conversely, in the INT8 implementation, the partial sum results are already summed into a single register, but there is a need to convert this number to a floating-point prior to adding the bias and applying ReLU. Unlike the FP16 implementation, which uses half precision throughout the computation, the INT8 implementation converts the integer to a single-precision floating-point number.

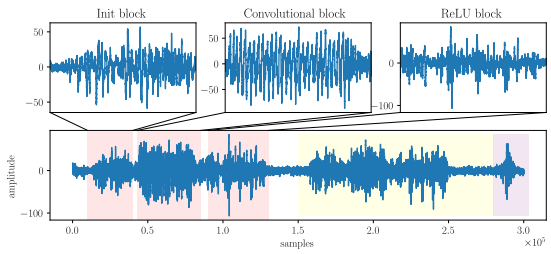


Figure 3: Raw trace of the whole operation on the GPU of Jetson Nano. The two convolutional layers (light pink and yellow) are clearly separated in the traces. Additionally, the CUDA device-to-host memory copy (light purple) is also clearly visible in the end of the trace.

Matching Instruction Block to Traces. Figure 3 shows the electromagnetic emanations of the Jetson Nano GPU during the execution of a CNN with two convolutional layers. Since each layer is mapped to a separate CUDA function call by the framework, it can be observed that there is a clear separation between layers. Additionally, the convolution results are copied back to memory, which is a separate CUDA call and is visible in the trace.

In addition, the three blocks of the implementations can also be identified. The first highlighted part shows the init block in the first layer. The second highlighted block corresponds to the convolutional block where the partial sums are computed. The third highlighted segment corresponds to the calculation of the bias addition and ReLU output, repeated four times for different sets of registers. Note that these instruction blocks are also separated by synchronization instructions, which are also visible in the trace as the amplitude of the EM signal drops close to 0 between the blocks.

4.2 Identifying Partial Sums

As no single GPU instruction can process an arbitrary number of arguments, the code typically processes these convolutions as a sequence of partial sums. An example of a naive way of computing Equation 1 is by sequentially computing partial sums s_j using the formula:

$$s_j = \sum_{i=1}^{j} w_i \cdot x_i = s_{j-1} + w_j \cdot x_j \qquad (j \leq p \cdot q) \qquad (3)$$

With this formula, if we assume we know s_{j-1} and x_j, we can guess w_j and use the guess to compute s_j. We then apply a leakage model to the computed s_j, e.g., the Hamming distance between s_{j-1} and s_j, and search for correlations between the model and observed leakage. A high correlation indicates that the guess of w_j is correct.

The challenge is that there are multiple ways in which the code can compute Equation 1. In particular, the implementation can change the order of computing the sum, and it can also use vector operations to combine multiple additions into a single operation. As our attack relies on knowing the previous partial sum to guess the weight, we must know how the sum is computed.

In this section, we delve into the implementations of convolutions with two reduced-precision implementations used by the TensorRT framework: INT8 and FP16. Reduced-precision implementations, such as these, are typically used during inference because they reduce both the latency and the memory use compared to single-precision implementations, which are typically used in the training phase. Our choice of INT8 and FP16 targets the larger and more challenging parameter sizes in reduced-precision techniques [30, 43, 45, 50, 50], and covers both integer and floating-point parameters. Here we describe the high-level design of the implementations. See Appendix A for further technical details.

FP16 Convolution. The FP16 implementation uses the `HFMA2` instruction, which performs two half-precision fused-multiply-adds in parallel. It operates on 32-bit registers, each holding two half-precision floating-point numbers. It first computes the products of two pairs of numbers in matching halves of two registers and then adds the results to the matching halves of a third register, which acts as an accumulator. This, basically, splits the convolution computation across the two channels, which are summed at the end. Our leakage model targets each partial sum (16 bits) that is written into this accumulator register.

INT8 Convolution. The INT8 implementations of convolutional and dense layers use the `IDP.4A` instruction to perform a 4-way dot product and accumulation operation, depicted in Figure 9. The instruction first multiplies the elements in matching channels in two registers. It then sums the results and adds them to a third register, which stores a signed 32-bit accumulator.

In many cases, not all four channels of the dot product contain data. Specifically, when the number of input channels is three, the channels of a single input point are convolved with the matching weights. On the other hand, when the input consists of a single channel, two input points are convolved during each operation. Values for channels that are not used are set to zero and do not affect the result of the instruction.

Unlike the `HFMA2` instruction, the sum depends on all of the weights that are convolved by the `IDP.4A` instruction. Thus, the attack needs to target all of the weights that are used by a single instruction.

4.3 Localizing Partial Sums

Section 4.1 demonstrates that we can identify the high-level operations in the trace, including the layer processing and the main steps of their computation. In Section 4.2, we show how we find the partial sums that leak specific weights. In this section, we complete the profiling and identify the trace locations that leak each partial sum and, consequently, the weights. Our approach employs TVLA [24] to find statistical evidence of leakage.

TVLA. To determine the time points in the traces where values of the partial sums s_j leak, we apply fixed vs. random TVLA to find statistical evidence for leakage. Specifically, we collect two sets of traces. In the "fixed" set, we set the target weight to a fixed non-zero value, all other weights to zero, and all of the input to fixed randomly chosen values and collect multiple traces of performing inference with the model. For the "random" set, we similarly collect multiple traces, but for each trace, we randomly select the value of the target weight. All other weights and inputs are set to the same fixed values as in the "fixed" set. We then compare the distribution of values of each time point across the two sets using Welch's t-test. As common when performing side-channel attacks, if the absolute t-value is above 4.5, i.e., $|t| > 4.5$, we mark the point as a potential leakage.[7]

Figure 4 shows the results of fixed vs. random TVLA for the first weight in the kernel in the first layer for FP16 convolution on the Jetson Nano. Leakage is clearly evident at the start of the convolutional block, around sample 25 000, where the t value goes above 4.5. Repeating the process for all of the weights of the kernel allows us to identify the time point for each weight at which they leak.

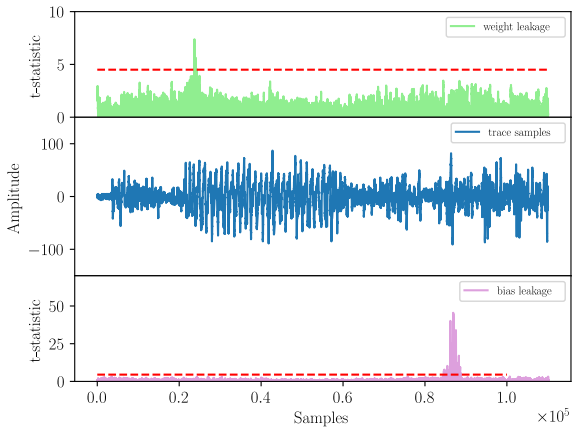


Figure 4: Result of fixed vs. random TVLA for the first weight (top) and the bias (bottom) in FP16 convolution with 30K and 37K traces, respectively. The middle depicts an example trace. The dashed red line indicates the 4.5 threshold.

To create the random set for fixed vs. random TVLA, we need to repeatedly modify the target weight to a random value. The TensorRT framework supports changing the weights of models, but a new CUDA context [4] has to be created every time the weights of a network are changed. This contrasts with the fixed case, where an application is not required to create a new CUDA context for every inference operation. However, as TVLA requires changing the weights, we cannot avoid this.

4.4 Bias Leakage

For the bias, the final partial sum c_{out} or the output of the activation c_f has to leak. Similarly, to detect leakage corresponding to the bias, we apply fixed vs. random bias TVLA.

Figure 4 also shows the results of TVLA for the bias of the kernel in the first layer for FP16 convolution. A much clearer leakage is present for the bias than for the weights in the convolution operation.

After establishing where parameters leak, with the help of elastic alignment, we use static alignment on the raw traces at these points as static alignment produced higher individual TVLA peaks as well as correlation for the attack.

4.5 CEMA Implementation in CUDA

In order to calculate correlation in CEMA, the covariances of populations have to be calculated, which can be done with two-pass algorithms. However, for large datasets, two-pass algorithms are inefficient as the algorithm makes two iterations on the dataset. Therefore, one-pass algorithms have been

[7] To verify that TVLA does not yield false positives, we check the corresponding intermediate-value correlations of a few weights to confirm the leakage. For example, when a computation followed Equation 3, we used $HD(s_{j-1}, s_j)$. All these experiments confirmed the TVLA results.

developed to estimate these statistics in large datasets [47]. In a one-pass algorithm, the statistics can be updated when new data points are added to the dataset. These algorithms also make it possible to combine the statistics from subsets of the dataset to estimate the whole dataset's statistics. This means that the statistics of each subset can be calculated in parallel, further speeding up the CEMA attack.

However, the attack has even more aspects that can be parallelized, such as the *candidate* and *sample* levels. There are publicly available multithreaded implementations such as the JlSCA library [8] for CPU, but these cannot fully parallelize the attack and become slow for large datasets. Since we have a large number of candidates with FP16 weights, we decided to implement CEMA for neural networks in CUDA, parallelized on three levels: dataset, candidate, and sample.

Our implementation in CUDA launches a three-dimensional grid of thread blocks with dimensions: $(candidates/2, chunks, samples) = (17765, chunks, 32)$. For our use case, the number of candidates and samples are fixed at 35530 and 32, respectively. Threads in the same warp work on the same two candidates (in parallel, due to double throughput with FP16) but correlate them with different samples. The implementation also parallelizes CEMA on a data set level, by setting $chunks > 1$. The number of chunks can vary, but setting it to 10 already gives good results in our use case with millions of traces. The optimal number may be lower or higher depending on the GPU hardware.

We benchmark the multithreaded JlSCA implementation on an AMD Ryzen 7950X CPU vs. our CUDA implementation on a 3080 Nvidia RTX GPU. Overall, the speedup compared to JlSCA is at least $\times 5$ and typically $\times 10$ when the dataset consists of millions of traces.

5 Parameter Extraction Results

In this section, we demonstrate the generality of our parameter extraction framework, shown in Figure 1, on a real-world CNN architecture by targeting the baseline EfficientNet [59] to extract a kernel from its first two convolutional layers. Note that none of the previous works have demonstrated parameter extraction from real-world CNN architectures.

In these experiments, after using the profiling information from Section 4, we use the *partial sums* to extract FP16 and INT8 weights on the Jetson Nano and Jetson Orin Nano, respectively. In addition, we also analyze the impact of batch size on the attack.

5.1 FP16 Parameter Extraction

Following [13], we restrict the search space of the FP16 weights to $[-5, 5]$, as most of the parameters of trained CNNs reside in this range. To verify this, we looked at the parameters of large real-world architectures, such as our target EfficientNetB0, trained on ImageNet. With 16-bit floats, there are 35 330 possible candidates in this range.

In the experiment, the parameters of the target architectures are initialized randomly. Observe that while we attack in an iterative fashion, the trace acquisition needs to be executed only once with random known inputs. We target the FP16 *partial sums* in each layer to extract weights and biases.

5.1.1 Convolutional Layer

In this experiment, we show the results of our framework by successfully extracting FP16 parameters from the first 2 convolutional layers of the baseline EfficientNet [59] architecture by targeting the partial sums in the layer.[8] Both the HW and HD leakage models prove to be exploitable in recovering weights and biases from the layers. Targeting the partial sums allows us to use a divide-and-conquer approach by reducing complexity, as only one FP16 weight must be extracted from a kernel at a time.

Weight Extraction. For the weights, we target the partial sums s_j of the 2D convolution and extract the weights of a kernel one by one. First, we are able to extract weights using the HW leakage model. As shown in Figure 5a and Figure 5b, we successfully recover the third weight of a kernel in the first layer. For the second layer, Figure 5e and Figure 5f show the key ranking and correlation for the second weight in the second layer. In addition, Figure 5g and Figure 5h show the key ranking and correlation for the third weight in the second layer, demonstrating that our attack extends to larger layers as well.

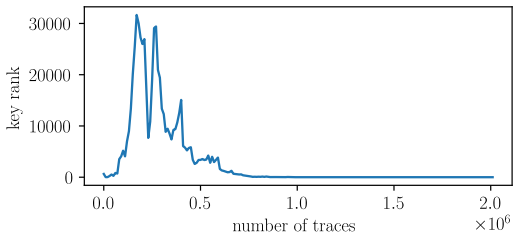
The $HD(s_{j-1}, s_j)$ leakage model targeting a register update can also be exploited to recover weights: Figure 5c and Figure 5d show the key rankings and correlations for the 9th weight in the first layer of the large architecture using HD. The attack works similarly for the other weights. As shown in Figure 5c, the convergence behavior for HD is different from that for HW as it starts to converge slower.

Bias Extraction. Similarly to the weights, we are also able to use HD leakage model to extract the bias: we targeted the register update from c_{sum} to c_{out}: $HD(c_{sum}, c_{out})$. Figure 6e and Figure 6f show the results for the bias in the first layer. The key rank drops quickly and converges to key rank 0 in 5 million traces. Overall, there is a significant variance in the number of required traces to recover individual weights and biases, but an upper bound of 20 million traces proves sufficient in our experiments.

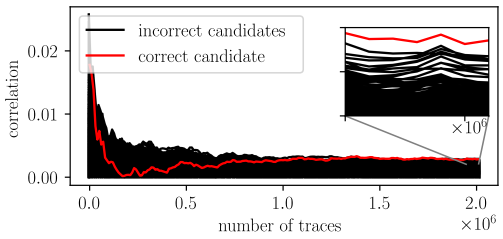
5.2 INT8 Parameter Extraction

For INT8 weights, we demonstrate our parameter extraction framework on the first convolutional layer with the same Ef-

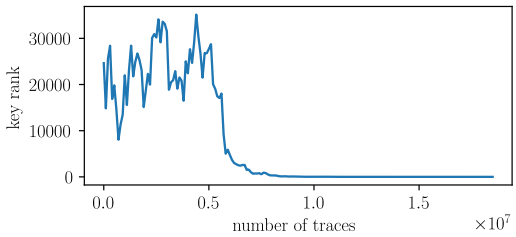
[8] The convolutional layers in this architecture do not have biases. Therefore, we alter the architecture so that the first layer has biases in the kernels to demonstrate the bias extraction on this architecture as well.



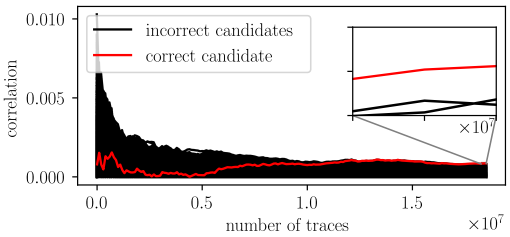
(a) Key rank vs. number of traces using *HW* for the third weight in the first layer with value of 0.8223.



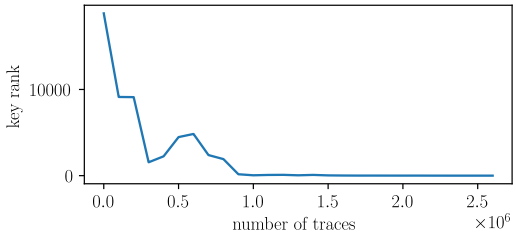
(b) Correlation vs. number of traces using *HW* for the third weight in the first layer with value of 0.8223.



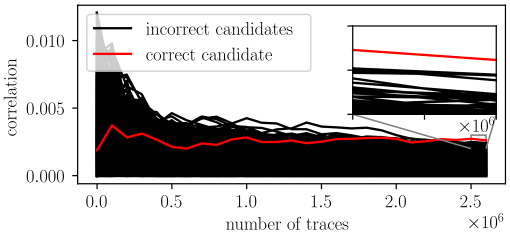
(c) Key rank vs. number of traces using *HD* for the ninth weight in the first layer with value of -0.7705.



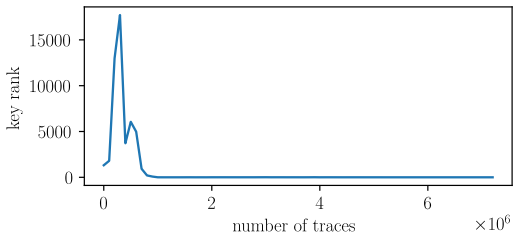
(d) Correlation vs. number of traces using *HD* for the ninth weight in the first layer with value of -0.7705.



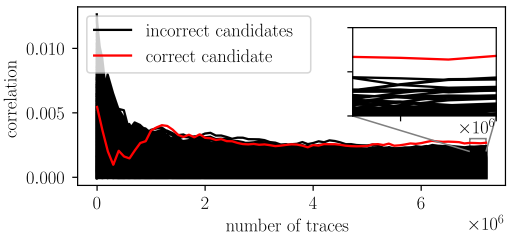
(e) Key rank vs. number of traces for s_j for the second weight in the second layer with the value of -0.5137.



(f) Correlation vs. number of traces for s_j for the second weight in the second layer with the value of -0.5137.



(g) Key rank vs. number of traces for s_j for the third weight in the second layer with the value of -0.6406.



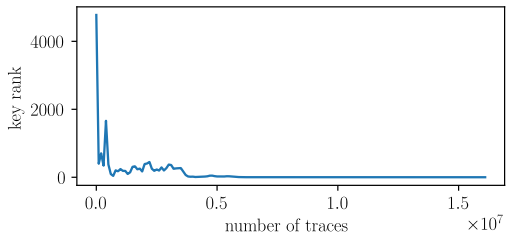
(h) Correlation vs. number of traces for s_j for the third weight in the second layer with the value of -0.6406.

Figure 5: Key ranks and correlations of the different FP16 weights in the first and second layer on the Jetson Nano.

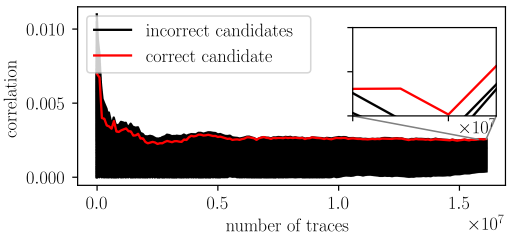
ficientNetB0 configuration as in the FP16 case, also with different batch sizes. Similarly, we target the partial sums in the INT8 convolution to extract the weights. In addition, we provide results on dense layer parameter extraction with INT8 weights.

5.2.1 Convolutional Layer

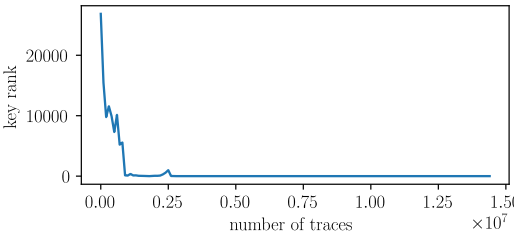
Weight Extraction. In this case, the partial sums can depend on multiple weights, depending on the number of input channels. In this experiment, the number of input channels to



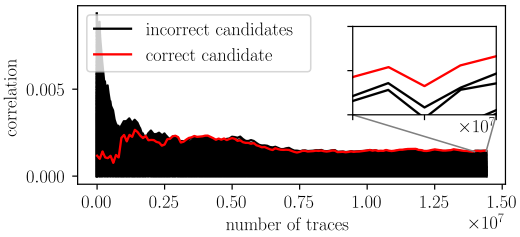
(a) Key rank vs. number of traces for ReLU.



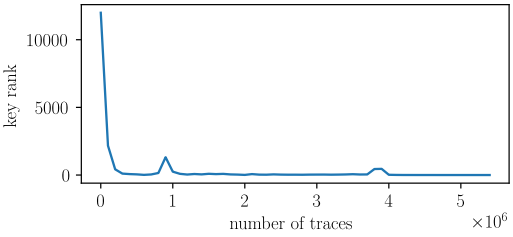
(b) Correlation vs. number of traces for ReLU.



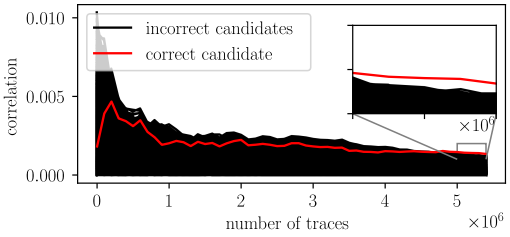
(c) Key rank vs. number of traces for c_{out}.



(d) Correlation vs. number of traces for c_{out}.



(e) Key rank vs. number of traces when c_{sum} is overwritten with c_{out}.



(f) Correlation vs. number of traces when c_{sum} is overwritten with c_{out}.

Figure 6: Key ranks and correlations of the FP16 biases with different leakage models on the Jetson Nano.

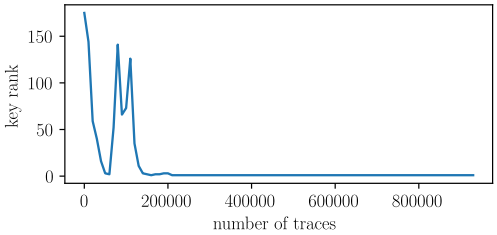
the first layer is one, so the partial sums depend on exactly one 8-bit weight. If the number of input channels is larger than 1, as is the case in subsequent layers, then the target intermediate values depend on multiple weights, increasing the complexity of the attack.

Similarly to the FP16 case, we extract the weights in a kernel one by one by targeting the partial sums in the convolution. Figure 7a and Figure 7a show the results for 4th weight in a kernel, while Figure 7c and Figure 7d show the results for the 5th weight in a kernel, respectively. In our experiments, on average, 300K traces were enough for the correct key candidates to reach key rank 0. Both the HW and HD leakage models are exploitable to recover individual weights.

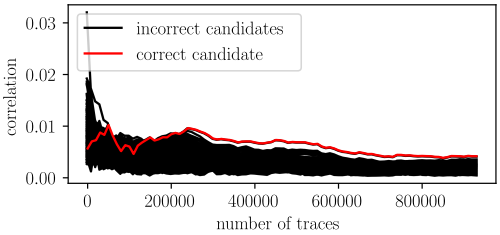
Impact of Batch Size. The batch size does influence the number of traces required to extract weights but not significantly, as 500K traces are sufficient to reach key rank 0. The difference between smaller and larger batch sizes is the number of executing threads. For instance, the first convolutional layer is executed with 512 and 7 232 threads for batch sizes of 1 and 16, respectively. Even though there are 14 times more threads for batch size 16, the number of executing threads physically in parallel is limited, e.g., the warp schedulers still only issue instructions for at most 32 threads at a time. Therefore, on smaller GPUs, the main impact of batch size might be only the linear increase in execution time. On the other hand, a larger batch size provides even more partial sums in one trace, and a horizontal attack [18] exploiting this might enhance the efficacy of the attack. We leave analyzing this for future work.

5.2.2 Dense Layer

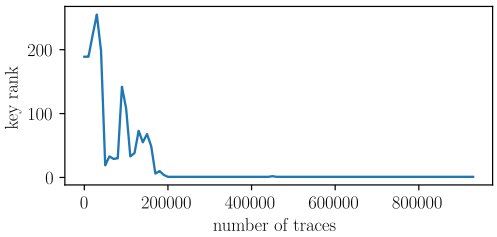
We present parameter extraction results of a neural network containing a single dense layer with INT8 weights running on the Jetson Orin Nano. In the experiments, our target dense layer has 512 nodes, and the input size s of the layer is 784. Therefore, each node has 784 weights associated with it. Our leakage modeling experiments for dense layers discovered that, similarly to convolutional layers, the HW and HD leakage models are both viable for mounting a successful pa-



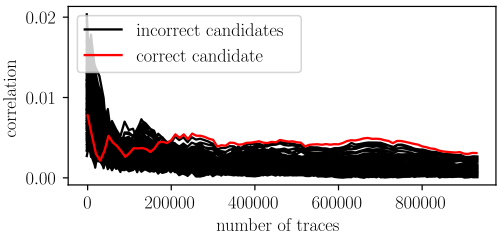
(a) Key rank vs. number of traces with HD leakage model for the fourth weight in the convolutional kernel.



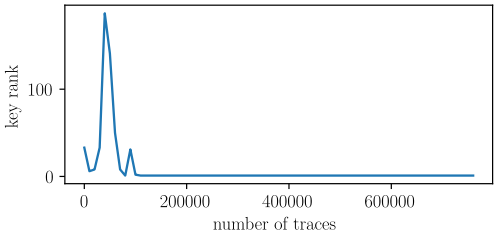
(b) Correlation vs. number of traces with HD leakage model for the fourth weight in the convolutional kernel.



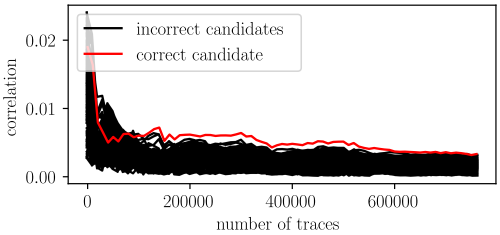
(c) Key rank vs. number of traces with HD leakage model for the fifth weight in the convolutional kernel.



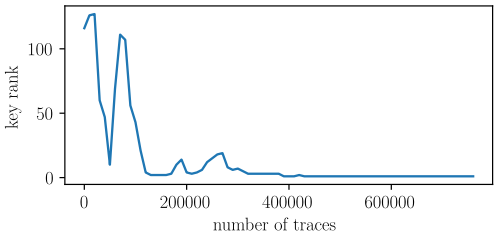
(d) Correlation vs. number of traces with HD leakage model for the fifth weight in the convolutional kernel.



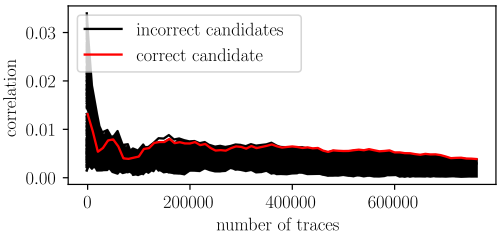
(e) Key rank vs. number of traces with HW leakage model for the 8th weight in the dense layer.



(f) Correlation vs. number of traces with HW leakage model for the 8th weight in the dense layer.



(g) Key rank vs. number of traces with HD leakage model for the 8th weight in the dense layer.



(h) Correlation vs. number of traces with HD leakage model for the 8th weight in the dense layer.

Figure 7: Key ranks and correlations of the different INT8 weights of convolutional and dense layers on the Jetson Orin Nano.

rameter extraction attack. However, these exploitable leakage models depend on multiple, not just one, 8-bit weights because the input size to dense layers is significantly larger than 4 in real-world applications. Therefore, most or all partial sum results depend on four weights, i.e., we face a complexity of 32 bits with the INT8 data type. To combat this issue, we apply the chosen-input attack mentioned earlier by setting three channels to 0 so that the registers that hold the input values contain only 1 non-zero input. Therefore, the final result will only depend on one 8-bit weight. Since all the nodes in the layer receive the same inputs, the chosen input attack does not require a new trace-set for every node separately. Therefore, the attack is *independent* of the size of the layer.

Figure 7e and Figure 7f show the results with HW leakage

model while Figure 7g and Figure 7h show the results with HD leakage model for the 8th weight in the first node. In our experiments, on average, 300K traces were enough for the correct key candidates to reach key rank 0, similar to the convolutional layer. In addition, both the HW and HD leakage models are exploitable to recover individual weights, which is unsurprising since the same instruction is used for the layers.

5.2.3 Attacking Deeper Layers.

In deeper layers with INT8 weights, each partial sum depends on four 8-bit weights as the number of input channels is larger than four. Generally, this increases the complexity of the attack to guessing four weights at a time, so 2^{32} candidates, which is computationally expensive. However, if the attacker has enough computing resources, the attack can still be mounted by attacking 32 bits.

However, the complexity can be reduced in two similar ways:

1. Collect traces with random inputs and choose only those where the inputs to deeper layers in specific channels are zero.
2. Collect traces with chosen inputs so deeper layers receive zero inputs in specific channels.

Random Inputs. This approach implies that the attacker has to collect significantly more traces to have 300K traces with the desired 0-value inputs in specific channels. However, the inputs in DNNs are usually centered around zero, and some of their quantized values are exactly 0. More importantly, the ReLU activation function greatly enhances sparsity in the inputs due to setting negative values to 0.

Chosen Inputs. In this approach, the attacker has to solve systems of linear equations to generate inputs to the first layer such that the inputs to deeper layers are 0, similarly to [25]. The number of collected traces is significantly less than that of random inputs, but whether this approach works for deeper layers in practice is still to be determined. Nevertheless, quantization and the ReLU activation function also enhance this approach as these factors make the solutions space larger.

5.2.4 Parameter extraction comparisons

The summary of parameter extraction of the target EfficientNetB0 architecture on different GPUs is presented in Table 2. For the Jetson Nano with FP16 parameters, the first two convolutional layers can be extracted with a complexity of 2^{16} for each parameter. However, a full model extraction is still computationally expensive as the model contains millions of parameters. In this case, the parameters required at most 20M traces for successful extraction. Observe that the number of traces should not increase even if we attack more layers because the attack does not make assumptions about layers' inputs.

	Jetson Nano	Jetson Orin Nano
weights data type	FP16	INT8
bias data type	FP16	FP32
# of extracted layers	2	1
# of extracted parameters	9564	288
bias extraction	✓	-
# of max req. traces	20M	3M

Table 2: Parameter extraction results for different GPUs for the EfficientNetB0 architecture.

For the Jetson Orin Nano with INT8 weights, we were only able to extract the weights of the first layer as the parameters in the second layer have a complexity of 2^{32} candidates due to how partial sums depend on 4 weights. In addition, the biases are represented as 32-bit floats that also provide a complexity of 2^{32} candidates. Although the number of required traces is at most 3M for the first layer weights, subsequent layers require most likely more as there are 2^{32} candidates. Overall, even with FP16 parameters, a full model extraction requires computing resources that are beyond our capabilities but not beyond a well-founded commercial or state attacker.

6 Discussion

6.1 Approaching Additional platforms

Our approach is demonstrated on CUDA-enabled GPUs, but we expect that our methodology is applicable to other platforms as well. An attack on another platform would also start with a profiling phase where the target architecture's implementation is reverse-engineered on a low level. This is necessary to identify and localize partial sums allowing for parameter extraction. After the profiling phase, the attacker is equipped with the appropriate leakage models and locations to extract the parameters. We expect that there would be some differences with respect to the exact intermediate values that would need to be targeted, but the approach should be similar.

6.2 Desktop/Datacenter GPUs

Against large GPUs, our attack can be extended and is likely to be more expensive depending on the target neural network's size. One key difference between the Jetsons and desktop/-datacenter GPUs is the number of streaming multiprocessors. Ideally, for a large GPU, an attacker would first locate all the SMs of the target GPU. Afterward, each SM could be scanned with an EM probe to see if there is any activity of interest. If the target neural network is extremely large and saturates all the SMs during inference, multiple probes may be used to cover all of them and collect traces in parallel for each SM. Therefore, the equipment and overall cost is higher the more

SMs the target GPU has.

6.3 Limitations

While we demonstrated parameter extraction on multiple GPUs, the coverage is still limited:

Concurrent Applications. GPUs are able to handle and schedule concurrently CUDA functions from multiple applications. This might introduce noise, but it depends on the required resources for each CUDA function. Suppose a CUDA function takes up most of the GPU resources (e.g., shared memory, registers, etc.). In that case, a different CUDA function will only be scheduled after all the thread blocks in the previous CUDA function finished execution [3]. Therefore, on GPUs with few SMs, this is less likely to be an issue as the GPU's resources are already saturated due to the large computational demand of DNN inference.

Bias Extraction in INT8 Implementations We demonstrated the extraction of biases from convolutional layers for FP16 implementations but not INT8 implementations. These implementations only use INT8 for the weights but not for the bias. The bias in these implementations has FP32 data type, which is beyond our computing capabilities to extract.

6.4 Mitigation

Traditional ways to contain electromagnetic emanation, such as proper shielding or introducing noise to decrease the Signal-to-Noise ratio, could alleviate the problem [42]. Specifically against parameter extraction, one of the possible countermeasures, which is also mentioned in the CSI-NN paper [13], is shuffling [61] the order of multiplications in the layers, which can make it significantly harder for an adversary to recover the weights. Additionally, masking [19, 48] can also decouple the side-channel measurements and the processed data. However, this comes at the price of execution speed, which might not be desired in real-time systems. Specifically for convolution, the registers containing the results of the partial sums can be initialized with the bias of the kernel instead of initializing them with zeros. This would prompt an adversary to mount a CEMA attack where the correct $b + w_1$ pair has to be recovered first. The complexity of this attack would be 32 bits due to 16 bits of complexity for the weight and bias separately in the FP16 case. However, in the INT8 case, the bias is a single-precision float, so it cannot be used to initialize the accumulator registers.

6.5 Related Work

To the best of our knowledge, no previous work has been able to extract the parameters of neural networks on GPU using physical side-channel. Previous works have demonstrated parameter extraction on microcontrollers and FPGAs using power or EM side channel, as shown in Table 3. In addition, these attacks were performed on neural networks with binary parameters [21, 53, 63], 8-bit parameters [13, 25, 37, 64] or 32-bit parameters [13, 31]. Our work demonstrates parameter extraction of 8- and 16-bit parameters. Furthermore, our work presents a CEMA attack on weights where the number of cores and the clock frequency at these cores operate are significantly larger than in related works. The large number of cores, with almost 1GHz clock frequency, presents a challenge in both the measurement and attack stages. Given that GPUs are the backbone of AI, it is of utmost importance to assess the resilience of GPU accelerated workloads against weight extraction attacks, a task our research addresses.

Author	Platform	Clock (MHz)	Side channel	Parameter datatype
Batina, et al. [13]	MCU	20, 84	EM	FP32
Dubet, et al. [21]	FPGA	24	Power	Binary
Yoshida, et al. [64]	FPGA	25	Power	INT8
Regazzoni, et al. [53]	FPGA	N/A[9]	EM	Binary
Yli-Mäyry, et al. [63]	FPGA	N/A[9]	EM	Binary
Li, et al. [37]	FPGA	25	Power	INT8
Joud, et al. [31]	MCU	100	EM	FP32
Gongye et al. [25]	FPGA	320	EM	INT8
BarraCUDA	**GPU**	**625**, **920**	EM	INT8, FP16

Table 3: Comparison with related work.

7 Conclusions

In this work, we analyzed the GPUs of Nvidia Jetson Nano and Nvidia Jetson Orin Nano, commonly chosen platforms for real-world neural network implementations, for resilience against side-channel attacks that aim to extract the weights of the target NN. First, we find multiple vulnerable points where the GPUs leak information about the parameters of the target DNN. Subsequently, we demonstrate the extraction of weights and biases of convolutional and dense layers. Overall, the neural network implementations of Nvidia's TensorRT framework are vulnerable to parameter extraction using EM side-channel attack despite the networks running in a highly parallel and noisy environment. Protecting their implementations in security or privacy-sensitive applications remains an open problem.

8 Acknowledgments

This research was supported by: an ARC Discovery Project number DP210102670; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; MV Impact Ai-SecTools (VJ02010010). In addition, this work

[9]The clock frequency is not disclosed in these attacks, but it is at most 800MHz as both attack XILINX ZYNQ chip [12].

was (in part) supported by Dutch Research Council (NWO) through the PROACT project (NWA.1215.18.014), TTW PREDATOR project 19782 and the CiCS project of the research programme Gravitation under the grant 024.006.037.

9 Ethics considerations

We notified Nvidia of the vulnerabilities found, and they acknowledged our findings. Nvidia recommends that users follow guidelines to prevent physical access and information leakage. In addition, we used our own equipment during the experiments in our laboratory and no team-member's well-being was adversely affected.

10 Open science

We provide code to run neural network models for each GPU device investigated in this work. We also provide the code used for elastic alignment. Lastly, we provide a CUDA implementation for running the attack and an example traceset to try. The artifacts are available at https://zenodo.org/records/14678147.

References

[1] https://www.langer-emv.de/en/product/mfa-active-1mhz-up-to-6-ghz/32/mfa-r-0-2-75-near-field-micro-probe-1-mhz-up-to-1-ghz/854. Accessed: 2022-01-25.

[2] https://www.langer-emv.de/en/product/rf-passive-30-mhz-up-to-3-ghz/35/rf-b-0-3-3-h-field-probe-mini-30-mhz-up-to-3-ghz/17. Accessed: 2023-03-25.

[3] https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. Accessed: 2022-11-30.

[4] Cuda Context. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#context. Accessed: 2022-09-30.

[5] CUDA programming model. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model. Accessed: 2022-09-30.

[6] cuobjdump. https://docs.nvidia.com/cuda/cuda-binary-utilities/#usage. Accessed: 2022-09-30.

[7] Jetso orin nano module. https://developer.nvidia.com/downloads/assets/embedded/secure/jetson/orin_nano/docs/jetson_orin_nano_ds. Accessed: 2024-03-15.

[8] Jlsca. https://github.com/Riscure/Jlsca. Accessed: 2022-09-30.

[9] NVIDIA Jetson Nano. https://developer.nvidia.com/embedded/jetson-nano-developer-kit. Accessed: 2022-09-30.

[10] SIMT architecture. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture. Accessed: 2022-09-30.

[11] Tegra X1 System-On-Chip. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf. Accessed: 2022-09-30.

[12] ZYNQ Data Sheet. https://docs.xilinx.com/v/u/en-US/ds187-XC7Z010-XC7Z020-Data-Sheet. Accessed: 2022-09-30.

[13] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI–NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security*, pages 515–532, 2019.

[14] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, pages 16–29, 2004.

[15] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Frontiers in Handwriting Recognition*, 2006.

[16] Łukasz Chmielewski and Léo Weissbart. On reverse engineering neural network implementation on GPU. In *AIHWS*, pages 96–113, 2021.

[17] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, pages 1251–1258, 2017.

[18] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *ICICS*, pages 46–61, 2010.

[19] Jean-Sébastien Coron and Louis Goubin. On Boolean and arithmetic masking against differential power analysis. In *CHES*, pages 231–237, 2000.

[20] Josef Danial, Debayan Das, Santosh K. Ghosh, Arijit Raychowdhury, and Shreyas Sen. Scniffer: Low-cost, automated, efficient electromagnetic side-channel sniffing. *IEEE Access*, 8:173414–173427, 2019.

[21] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. In *HOST*, pages 197–208, 2020.

[22] Fürkan Elibol, Uğur Sarac, and Işin Erer. Realistic eavesdropping attacks on computer displays with low-cost and mobile receiver system. In *EUSIPCO*, pages 1767–1771, 2012.

[23] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Trans. Syst. Sci. Cybern.*, 5(4):322–333, 1969.

[24] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[25] Cheng Gongye, Yukui Luo, Xiaolin Xu, and Yunsi Fei. Side-channel-assisted reverse-engineering of encrypted DNN hardware accelerator IP and attack surface exploration. In *IEEE S&P*, 2024.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[27] Zhang Hongxin, Huang Yuewang, Wang Jianxin, Lu Yinghua, and Zhang Jinling. Recognition of electromagnetic leakage information from computer radiation with SVM. *Computers & Security*, 28(1-2):72–76, 2009.

[28] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[29] Peter Horvath, Lukasz Chmielewski, Leo Weissbart, Lejla Batina, and Yuval Yarom. CNN architecture extraction on edge GPU. In *AIHWS*, 2024.

[30] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *NeurIPS*, 2016.

[31] Raphaël Joud, Pierre-Alain Moëllic, Simon Pontié, and Jean-Baptiste Rigaud. A practical introduction to side-channel extraction of deep neural network parameters. In *CARDIS*, pages 45–65, 2022.

[32] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[33] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113, 1996.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *NeurIPS*, 25:1097–1105, 2012.

[35] Markus G. Kuhn and Ross J. Anderson. Soft Tempest: Hidden data transmission using electromagnetic emanations. In *International Workshop on Information Hiding*, pages 124–142, 1998.

[36] Nikolay Laptev, Jason Yosinski, Li Erran Li, and Slawek Smyl. Time-series extreme event forecasting with neural networks at uber. In *ICML*, pages 1–5, 2017.

[37] Ge Li, Mohit Tiwari, and Michael Orshansky. Power-based attacks on spatial DNN accelerators. *ACM Journal on Emerging Technologies in Computing Systems*, 18(3):1–18, 2022.

[38] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[39] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International journal of computer vision*, 128(2):261–318, 2020.

[40] Zhuoran Liu, Niels Samwel, Léo Weissbart, Zhengyu Zhao, Dirk Lauret, Lejla Batina, and Martha Larson. Screen gleaning: A screen reading TEMPEST attack on mobile devices exploiting an electromagnetic side channel. *NDSS*, 2021.

[41] Henrique Teles Maia, Chang Xiao, Dingzeyu Li, Eitan Grinspun, and Changxi Zheng. Can one hear the shape of a neural network?: Snooping the GPU via magnetic side channel. In *USENIX Security*, pages 4383–4400, 2022.

[42] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[43] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 formats for deep learning. arXiv preprint arXiv:2209.05433, 2022.

[44] Amir Moradi, Bastian Richter, Tobias Schneider, and François-Xavier Standaert. Leakage detection with the x2-test. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 209–237, 2018.

[45] Sharan Narang, Gregory Diamos, Erich Elsen, Paulius Micikevicius, Jonah Alben, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *ICLR*, 2017.

[46] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2):604–624, 2020.

[47] Philippe Pierre Pébay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Sandia Report SAND2008-6212, Sandia National Laboratories, 2008.

[48] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Eurocrypt*, pages 142–159, 2013.

[49] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang, and Tara Sainath. Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing*, 13(2):206–219, 2019.

[50] Jerry Quinn and Miguel Ballesteros. Pieces of eight: 8-bit neural machine translation. In *NAACL-HLT (3)*, pages 114–120, 2018.

[51] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and countermeasures for smart cards. In *E-smart*, pages 200–210, 2001.

[52] Syama Sundar Rangapuram, Matthias W. Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. *NeurIPS*, 2018.

[53] Francesco Regazzoni, Shivam Bhasin, Amir Alipour, Ihab Alshaer, Furkan Aydin, Aydin Aysu, Vincent Beroulle, Giorgio Di Natale, Paul D. Franzon, David Hély, Naofumi Homma, Akira Ito, Dirmanto Jap, Priyank Kashyap, Ilia Polian, Seetal Potluri, Rei Ueno, Elena Ioana Vatajelu, and Ville Yli-Mäyry. Machine learning and hardware security: Challenges and opportunities. In *ICCAD*, pages 141:1–141:6, 2020.

[54] Alaa Sagheer and Mostafa Kotb. Time series forecasting of petroleum production using deep lstm recurrent networks. *Neurocomputing*, 323:203–213, 2019.

[55] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

[56] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In *CHES*, pages 495–513, 2015.

[57] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[58] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[59] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *ICML*, pages 6105–6114, 2019.

[60] Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. Improving differential power analysis by elastic alignment. In *CT-RSA*, pages 104–119, 2011.

[61] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *Asiacrypt*, pages 740–757, 2012.

[62] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, pages 2003–2020, 2020.

[63] Ville Yli-Mäyry, Akira Ito, Naofumi Homma, Shivam Bhasin, and Dirmanto Jap. Extraction of binarized neural network architecture and secret parameters using side-channel information. In *ISCAS*, pages 1–5, 2021.

[64] Kota Yoshida, Takaya Kubota, Shunsuke Okura, Mitsuru Shiozaki, and Takeshi Fujino. Model reverse-engineering attack using correlation power analysis against systolic array based neural network accelerator. In *ISCAS*, pages 1–5, 2020.

A Convolution implementation details

Listing 1: FP16 convolution code snippet

```
HFMA2      R0, R94, R109, R0;
HFMA2      R3, R92, R108, R3;
LDS.U.128 R116, [R88+0x210];
```

In this section, we show how the computation of partial sums differs based on the representation of the parameters and how this influences the attack. Specifically, the FP16 and INT8 data types have their own specialized instructions where the GPU registers are akin to vector registers.

FP16 convolution. As demonstrated in Listing 1, FP16-based convolution uses the `HFMA2` instruction, which performs two half-precision fused-multiply-adds in parallel. As illustrated

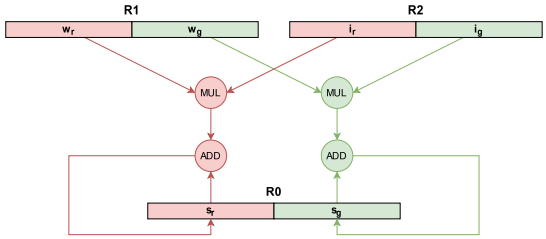


Figure 8: HFMA2 instruction operation with two input channels.

in Figure 8, the instruction takes three input registers, each is a two-lane vector. It multiplies the values in the matching channels of two registers, adds the result to the matching lanes of the third, and stores the result in the output register. In all instances we have seen, a single register is used as an accumulator, where the multiplication result is added to it. Our leakage model targets each partial sum (16 bits) that is written into this accumulator register.

Listing 2: INT8 convolution code snippet

```
IDP.4A.S8.S8 R62, R71, R83, R62;
IDP.4A.S8.S8 R59, R69, R82, R59;
LDS.128      R64, [R97+0x200];
```

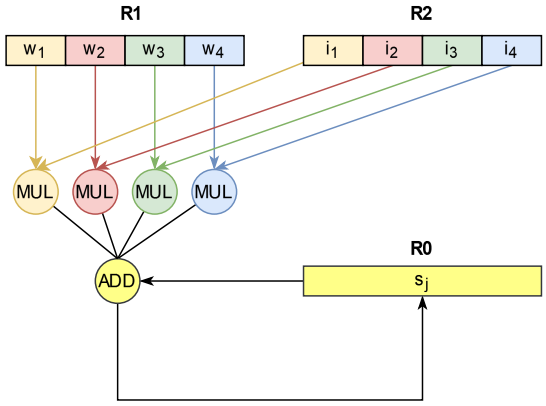


Figure 9: IDP.4A instruction operation with 4 input channels.

INT8 convolution. Listing 2 shows an example of code used in the INT8 implementation. The implementation uses the `IDP.4A` instruction to perform a 4-way dot product and accumulate operation, depicted in Figure 9. The instruction first multiplies the elements in matching lanes in two registers. It then sums the results and adds them to a third register, which stores a signed 32-bit integer. As in the FP16 case, in all uses we have seen, `IDP.4A` uses a single register as input and output numbers.

To extract weights, our leakage model targets the 32-bit partial sum (s_c in Figure 9) stored in the accumulator register. As the result depends on all four input lanes, the complexity of the CEMA attack grows to up to 32 bits. In practice, the code often uses less than four lanes. Specifically, when processing greyscale images, convolutions tend to use only one lane, whereas when processing color images, convolutions use three lanes, one for each color channel.

Using the INT8 representation on the Orin Nano, Tensor Core implementations use the `IMMA` (Integer Matrix Multiply and Accumulate) instruction. `IMMA` works on a warp level, meaning the threads in the warp must cooperate, but its execution is similar to `IDP.4A` on a thread level. The implementations we experiment with in this paper do not use the `IMMA` instruction. However, the techniques we develop for parameter extraction are directly applicable to it as well.

Dense layer. The implementation of dense layers follows the same design as convolutional layers. In these implementations, each accumulator register holds the partial sums of the weighted sum of a node in the layer. Similarly to INT8 convolutional implementations, the dense layer implementation uses the `IDP.4A` instruction to calculate the partial sums. Consequently, as in the convolutional, there is a need to guess 32 bits, except in edge cases, where some channels are set to zero.