AI Attacks AI: Recovering Neural Network Architecture from NVDLA Using AI-Assisted Side Channel Attack

NAINA GUPTA, Nanyang Technological University, Singapore, Singapore
ARPAN JATI, Nanyang Technological University, Singapore, Singapore
ANUPAM CHATTOPADHYAY, Nanyang Technological University, Singapore, Singapore

During the last decade, there has been a stunning progress in the domain of Artificial Intelligence (AI) aided by highly trained Machine Learning (ML) models. Such models are valuable Intellectual Property (IP) and, therefore, have been subjected to various model recovery attacks. In this work, we study the vulnerabilities of commercial, open-source accelerator NVDLA and present the first successful model recovery attack. For this purpose, we used power and timing information from the side-channel leakage of convolutional neural networks (CNN) models to train CNN-based attack models. Utilizing these attack models, we demonstrate that even with a highly pipelined architecture, multiple parallel execution in the accelerator along with Linux OS running tasks in the background, recovery of number of layers, kernel sizes, output neurons and distinguishing different layers, is possible with very high accuracy. This is also the first work to show the impact of differences in hyperparameters on the power traces.

Our solution is fully automated, AI-based, and portable to other hardware neural networks, thus presenting a greater threat toward IP protection. Using LeNet as the target victim model, we demonstrate an accuracy of more than 95% in recovering various parameters. This study presents a serious practical threat, in the form of side-channel attack, towards complex commercial architectures. Furthermore, we show that AI-guided attack significantly boosts the attacker capability.

CCS Concepts: • **Hardware** → *Hardware accelerators*; • **Computing methodologies** → *Artificial intelligence*; • **Security and privacy** → **Hardware reverse engineering**; **Side-channel analysis and countermeasures**.

Additional Key Words and Phrases: neural network accelerators, reverse engineering, NVDLA, CNNs, side-channel attacks, timing attacks, SPA, AI

1 Introduction

The growing adoption of Artificial Intelligence (AI) in various application segments has created a huge demand for efficient machine learning (ML) accelerators. Embedded ML accelerators represent a significant domain of such accelerators, which are integrated with Internet-of-Things (IoT) platforms to drive edge intelligence. Physical access to such platforms is not uncommon, and therefore various forms of (semi-)invasive attacks present a serious threat to the operation of embedded ML accelerators. In recent times, this has caught the attention of researchers, with multiple results clearly demonstrating the feasibility of such attacks for various platforms and different attack objectives.

The reported attacks can be broadly classified in terms of the attack objectives. First, when the attacker intends to recover the input data. This is demonstrated in [39] and [33] through power side-channel information. Second, when the attacker intends to disrupt the outcome of the Neural Network (NN) by combining adversarial attacks with active side-channel attack [13]. Third, also the focus of the current work is when an attacker intends to

Authors' Contact Information: Naina Gupta, Nanyang Technological University, Singapore, Singapore, Singapore; e-mail: NAINA003@e.ntu.edu.sg; Arpan Jati, Nanyang Technological University, Singapore, Singapore, Singapore; e-mail: arpan.jati@ntu.edu.sg; Anupam Chattopadhyay, Nanyang Technological University, Singapore, Singapore, Singapore; e-mail: anupam@ntu.edu.sg.

ACM 1558-3465/2025/4-ART
https://doi.org/10.1145/3731560

reverse engineer the NN through side-channel information. Further and overlapping taxonomies of attacks on embedded NN can be defined in terms of the types of NN (e.g., binarized, convolutional neural network), kinds of attack (e.g., passive, active, remote).

Despite the impressive body of work in recent times, there remain few hurdles towards deciphering the complete operations of practical embedded neural networks. Nearly all of the previous works have targeted embedded microcontrollers on which the neural network is being executed. It is comparatively easier to recover secret information from microcontrollers. This is mainly because every operation is executed sequentially in the case of microcontrollers. These settings considerably simplifies the problem at hand and is not always representative of a practical setup. One of the key requirements for a successful side-channel attack is the precise target and noise-free traces. Common optimization strategies used for FPGAs, such as pipelining and parallel execution, make side-channel attacks quite difficult. This is also noted in a recent paper along this line of research [27], which states that the parallel execution of multiple NN operations on an FPGA/ASIC platform makes the side-channel attack considerably harder. This is one of the reasons the current literature lacks analysis and possible attacks on hardware neural network accelerators.

In addition, many of the recent works in this direction focused on Binary Neural Network (BNN) architectures for FPGA platforms such as in [10, 41, 45]. Apart from this, few works such as in [4, 12] focused on the general-purpose acceleration feature of GPUs and not the embedded NN accelerators, which can be found in some recent GPUs. As GPUs have a very different architecture, they cannot be directly compared with our work. To the best of our knowledge, no prior work has yet investigated CNN architectures on hardware neural network accelerators using power traces. Furthermore, the analysis and evaluation of open-source commercial accelerators is yet to be explored for side-channel leakage. It is crucial to investigate the security of commercial accelerators due to their wide deployment and support of multiple network architectures compared to a custom HLS based accelerator. This is exactly what we address in this work.

Related Work. Side-channel attacks are well known in the literature for recovering secret keys from cryptographic algorithms. It was only recently that researchers have started exploiting side-channel leakage for recovering neural network architecture, weights, or inputs. For instance, Hua et. al. [14] used off-chip memory access information to recover the network structure. For this purpose, the authors implemented a CNN model with a hardware Trojan on a custom Vivado HLS based hardware accelerator. The authors demonstrated how memory access patterns can leak information about network structure even when off-chip memory data is encrypted. Later, Batina et al. [1] explored recovery of neural network structure and weights in a grey-box setting targeting microcontrollers using timing and electromagnetic (EM) side-channel measurements. The attack was successfully shown on two modern microcontrollers which are commonly used in pervasive applications. This is followed by the work in [46] where the authors used EM and a margin-based adversarial attack to recover the structure and weights for a BNN accelerator. Specifically, the authors targeted to recover substitute models using carefully crafted malicious examples to fool the system. In addition, the authors performed experiments on models such as LeNet, AlexNet, etc. and showed that they are able to build substitute models with good accuracy. In another attempt Maji et al. [27] utilized timing and Simple Power Analysis (SPA) techniques to recover inputs and models for different precisions such as fixed point, floating point, and binary NNs. Further, they demonstrated their attacks on multiple microcontroller-based devices. The work by Yoshida et al. [45] demonstrated weight recovery around a custom NN FPGA implementation based on systolic array. Another interesting attack to recover the architecture of neural networks is demonstrated in [48]. The authors utilized a remote power side-channel attack technique on a Zedboard. For this purpose, they implemented on-chip RO-based (Ring-Oscillators) power monitors to capture power profiles. The authors in [17] evaluated the open source library NNoM (Neural Network on Microcontroller) with the ARM CMSIS-NN library as the back-end). The work showed different power profiles corresponding to different NN operations when executed on a 32-bit microcontroller. For an excellent and timely survey of physical side-channel attacks on embedded neural networks, the readers may refer to [3, 31, 43].

Table 1. Overview of state-of-the-art. Modification includes additional circuit or Trojans.

Attack	Network Type	Physical Target	Attack Type	Remote	Automated	Scalable	Modifications*	Limitations
Hua et. al. [14], 2018	CNN	Custom self designed HLS based accelerator	Memory access patterns	×	×	×	✓	Hardware Trojan utilized to access memory trace. Target platform not publicly used, attack depends on the ability to control pruning threshold.
Batina et. al. [1], 2019	MLP, CNN	Atmel ATmega328P, ARM Cortex-M3 Microcontrollers	Timing & EM	×	×	✓	×	Hyper-parameters such as kernel size, stride, padding etc. not targeted.
Yu et. al. [46], 2020	BNN	Custom self designed HLS based accelerator	EM & Margin based Adversarial training	×	×	×	×	Target platform not publicly used. Guessed multiple parameters resulting in multiple NN candidates.
Maji et. al. [27], 2021	Low level operations like RELU, MAC, Multiply etc.	Microcontrollers such as ATmega328P, ARM Cortex-M0+ & custom RISC-V chip	SPA & Timing	×	×	×	✓	Disabled peripherals such as interrupt controllers, serial communication interfaces, etc. used in commercial microcontrollers.
Yoshida et. al. [45], 2020	Systolic Array	Custom wavefront array based implementation.	CPA	×	×	×	-	Utilized a simulated setup of just the systolic array instead of a full setup. Strong assumptions regarding model architecture.
Won et. al. [42], 2021	CNN	Raspberry Pi	Timing Templates	×	×	×	×	Distinguishing attack applicable only on known models.
Won et. al. [40], 2021	CNN	Raspberry Pi	Cold-boot attack	×	×	×	×	Not easily ported to other platforms.
Won et. al. [41], 2021	BNN	Intel NCS	CPA	×	×	×	×	Partial weight recovery with strong assumptions.
Dubey et. al. [10], 2020	BNN	Self designed BNN inference hardware accelerator	CPA	×	×	×	×	Target platform not publicly used. Accelerator is very specific to their trained network model.
Zhang et. al. [48], 2021	CNN	Custom model inferencing in hardware	RO-based power monitoring	✓	✓	✓	✓	Trace collection strategy limited to multi-tenant cloud-based FPGA platforms.
Joud et. al. [17], 2023	MLP, CNN	ARM Cortex-M7 micro-controller	EM	×	×	×	×	Hyper-parameters such as type, stride, padding etc. related to pooling layer are not targeted. Further, no evaluation corresponding to recovery of activation function is provided.
Yan et. al. [44], 2023	CNN	NVDLA	Sequence to sequence problem formulation and TDC based power collection	✓	✓	✓	✓	Trace collection strategy using a TDC (time to digital converter) is limited to FPGA-based setups for multi-tenant cloud-based FPGA platforms or hardware trojans.
Kurian et. al. [24], 2025	MLP, CNN	Google Edge TPU	Online template-building model attack	×	✓	✓	×	Weights and inputs not targeted. Requires approximately three hours for run-time computation and recovery for every layer.
This work	CNN	NVDLA	AI-assisted SCA	×	✓	✓	×	Weights and inputs not targeted.

All of the previous works targeted either microcontroller devices or a custom Vivado HLS based accelerator. There are many commercial accelerators as well, which are being used extensively for edge computing such as Google TPU [18], Intel NCS [15] and NVIDIA's NVDLA [36]. With regard to Intel NCS, three works have been published in the literature so far. The first work [42] developed execution time templates using a kernel density estimator (KDE) corresponding to Resnet and the VGG family of models to find the model used. This is later followed by a cold-boot based attack in [40]. In this case, the authors targeted to recover the model by freezing the RAM on Raspberry Pi. In both of the mentioned works, the authors utilized Raspberry Pi as the target for attack, with Intel NCS being used for inferencing purposes. The first attempt on an NN ASIC was made on an Intel NCS in [41]. The authors targeted recovery of model weights using an EM side-channel attack for a BNN model. Using CPA, the authors demonstrate some leakage in correlation with weights, but full recovery was not possible. The authors further state that the execution of NN inference is hard to distinguish in the captured trace, and model recovery was left as future work. A recent work by Yan et al [44] targeted NVDLA through remote power

trace collection and a sequence-to-sequence problem formulation. In order to mount the attack, the attacker must first deploy the proposed TDC (time-to-digital converter) circuit on the same FPGA and then collect the readout from this sensor. This attack strategy is completely different from our work. In our case, the attacker only needs to collect the power consumption traces. Further, as the attack is performed using sequence-to-sequence formulation, the hyperparameters were not targeted at a granular level. Whereas our attack model is more flexible with respect to different kernel sizes, output neurons, stride/padding sizes etc. In addition, the authors did not show any results corresponding to the recovery of stride and padding sizes. Furthermore, their trace collection strategy using a TDC is limited to FPGA-based setups for multi-tenant cloud-based FPGA platforms or hardware trojans. In comparison, our work is more general and applicable for multiple platforms, albeit requiring physical access. A recent work by Kurian et. al. [24] demonstrated successful recovery of neural network architecture on Google Edge TPU. The authors utilized the online template-building attack strategy rather than offline pre-trained ML-based models for recovery. Further, this is the first work to target non-sequential networks as well. For easier comparison, Table 1 summarizes the state of the art. One can note that all the attack techniques can be broadly classified into two categories: NN architecture recovery and input and weight recovery. Both of these require significantly different attack strategies and implementations. In this work, we targeted architecture recovery instead of input and weight recovery.

Contribution. As can be seen from Table 1 the current literature lacks an in-depth evaluation of widely deployed commercial NN hardware accelerators. In this work, we explore the possibilities of reverse engineering NN models from the NVIDIA open-source accelerator NVDLA, which is available in multiple NVIDIA Jetson Xavier platforms. We demonstrate successful reverse engineering of neural network architecture including number of layers, type of layers, etc. Further, this is the first work to demonstrate recovery of hyperparameters such as stride size, kernel size, padding size, etc. from a hardware accelerator. For this purpose, we first ported and integrated NVDLA to a Microblaze based system suitable for execution on a side-channel evaluation platform. In our attack, we used AI-assisted power analysis and timing side-channel attacks.

To the best of our knowledge, this is the first work analyzing the vulnerabilities of NVDLA in a practical and realistic setting. We demonstrate successful recovery of neural network parameters even from a deeply pipelined accelerator which allows parallel execution and runs alongside an OS. We effectively show that the leakage corresponding to different parameters is distinctively visible in the power traces and hence, can be directly exploited to recover any CNN model. Using LeNet as the target victim model, we demonstrate a very high accuracy of more than 95% in recovering different parameters.

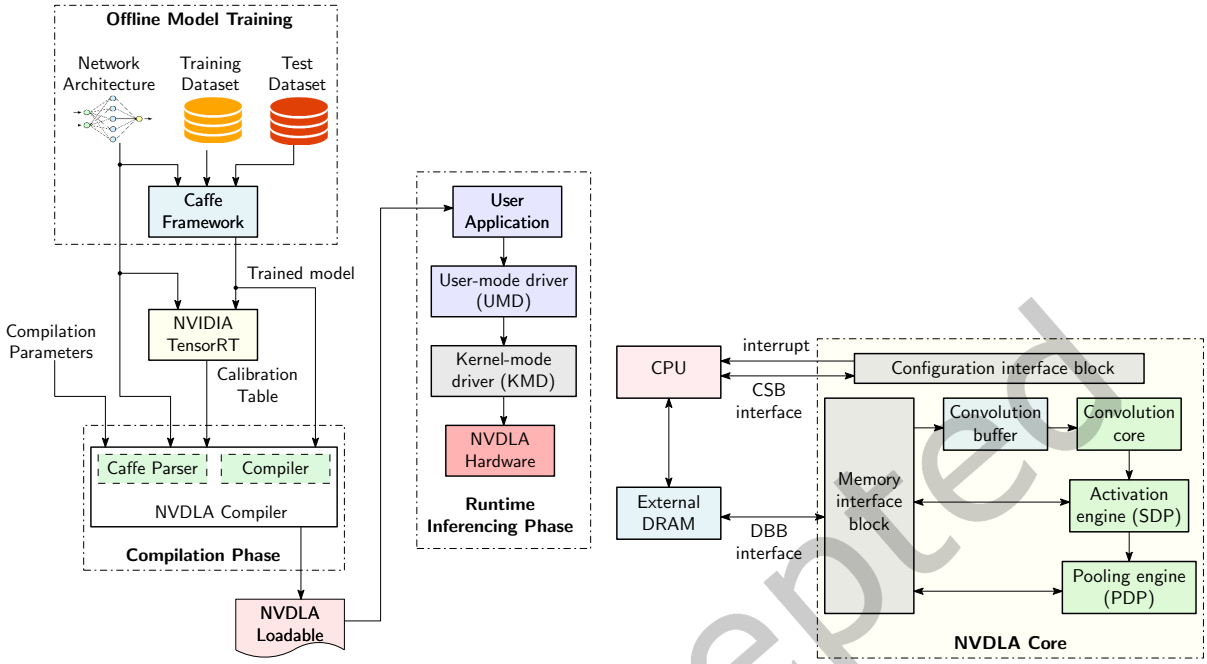
Organization. The paper is organized as follows. Section 2 provides the necessary preliminaries regarding the side-channel attack techniques (SPA and timing), the overall NVDLA architecture and platform flow and threat model. In Section 3, we present an in-depth analysis of the experimental setup, its challenges, and how we resolved them. This is followed by the model extraction discussion in Section 4. The results and evaluation of our attack are presented in Section 5 followed by some discussion in Section 6. We finally conclude the paper in Section 7.

2 Preliminaries

2.1 NVDLA Architecture

In this section, we briefly discuss the NVDLA architecture consisting of its software stack as well as the hardware architecture. The interested readers are referred to [36] for more in-depth details about the architecture.

Fig. 1a shows the overall flow of the NVDLA platform. NVDLA natively supports the Caffe framework only. Hence, the model is trained using Caffe. This is followed by generating the calibration table using NVIDIA TensorRT. This is required for quantization. NVDLA's software stack consists of a compiler which is used to parse the caffemodel and generate a loadable file. This loadable file consists of hardware understandable sequence and



(a) Overall platform flow (b) NVDLA core

Fig. 1. Hardware Architecture

parameters related to neural network model. The runtime environment of NVDLA consists of a user-mode driver and a kernel-mode driver. The two drivers act as portability layers between the user application and the NVDLA hardware. The user-mode driver is responsible for loading the network into main memory, parsing and preparing the input and output tensors, etc. Whereas, the kernel-mode driver is responsible for scheduling the different layers onto hardware.

The hardware architecture for the NVDLA core is shown in Fig. 1b. As can be seen in the figure, NVDLA has a dedicated unit that is used to perform different operations required in a neural network. The convolution core is used to perform the MAC operations required for a convolution and a fully connected layer. The single-point data processor (SDP) unit is used for bias addition as well as for the activation layer. Similarly, the Planar Data Processor (PDP) engine is used for pooling operations. The CSB interface is used by the kernel-mode driver to configure and schedule the required tasks for execution. For simplicity, we have shown the units that are available in the *nv_small* configuration. The *nv_large* or *nv_full* configuration further supports more features/units such as batch normalization, a dedicated SRAM interface, etc.

2.2 Side-Channel Attacks

The seminal work by Paul Kocher in 1996 [22] led to the development of a new form of attack known as Side-channel Attacks (SCA). These attacks focus on the implementation of a cryptographic algorithm instead of its mathematical structure to extract secret information (for instance, the key). Over the years, it has been shown that any device while performing cryptographic operations leaks data in the form of power consumption [7, 21, 32], electromagnetic emanations [2, 5, 11, 29], timing information [6, 23], etc. Since 1996, a lot of contribution has

been made in this field. Recently, practical attacks have been demonstrated on ML accelerators allowing the attackers to extract NN model architecture, its parameters such as weights, input, etc. and hence, are considered to be a serious threat.

2.2.1 Simple Power Analysis. The basic principle of a power analysis attack is to analyze the power consumption of a device when performing cryptographic operations. A power trace is the current signature produced when a device performs an operation. There are numerous ways in which power traces can be analyzed, and the relationship between secret data and power consumption can be exploited. One of them is Simple Power Analysis (SPA) [28, 30]. SPA usually requires a single trace, and the attack works by visual inspection or template-based analysis of the power trace.

2.2.2 Timing Attacks. Timing attacks exploit the execution time of any operation. The technique has been extensively applied to recover secret information depending on execution time. For instance, memory accesses with cache hit-or-miss, conditional branches, etc. result in varied execution times, which can be linked with the sensitive data. Such attacks necessitated the need for so-called constant-time implementations. In our study, we demonstrate how timing attacks can be employed to recover different parameters of a neural network.

2.3 Threat Model and Attacker's Motivation

In this paper, we assume that the attacker has physical access to the device with NVDLA being used for inferencing. In addition, the attacker can capture traces during execution and observe the power and timing patterns for the different operations. Note that the attacker only has access to the input, output, and side-channel traces. The attacker does not know anything about the neural network architecture during the attack phase. This knowledge is only available during the profiling phase.

The main target of the attacker is to recover the network structure by identifying different layer types and possible recovery of hyperparameters such as kernel size, number of filters, etc. One possible motivation behind the attack is IP stealing. Typically, it requires a lot of resources in terms of both computational power and time to train a neural network customized for a specific application by fine-tuning the parameters. A motivated attacker can attempt to recover the network. With the structure and hyperparameters being available, the training could be accomplished with a smaller data set and shorter duration of time, thereby gaining monetary benefits or market advantage. Further, the recovered model could also be useful as the first step for Generative Adversarial Network (GAN)-based model recovery. The kind of model stealing attack, where partial knowledge of the model is available is denoted as grey-box model stealing attack. In several recent works [19, 20, 37, 47], it has been shown that grey-box attack techniques offer an edge over black-box attack. This includes model recovery and general model inversion attack with fewer queries. We believe that the technique presented by us can complement such algebraic attacks as well.

3 Experimental Setup and Challenges

3.1 NVDLA integration

The first obstacle to analyze the security of NVDLA using side-channels is a working setup on a side-channel evaluation platform. In our experiments, we chose to use SASEBO-GIII as the target platform. It uses a Xilinx XC7K160T FPGA as the cryptographic FPGA and has 162,240 logic cells. Of the three configurations - *nv_small*, *nv_large* and *nv_full* provided by NVIDIA, the *nv_small* configuration takes about 90k logic cells. Thus, it is the only configuration that can fit in the evaluation platform along with the logic for a Microblaze based system. Hence, we integrated *nv_small* in our attack setup. The DDR3 based RAM is shared between Linux and NVDLA by mapping specific regions in the device tree. The hardware source code is available at (https://github.com/nvdla/hw/tree/nv_small) and the software is available at (https://github.com/nvdla/sw). We

faced many challenges, as discussed below, in order to successfully develop the set-up. These changes are necessary for any Microblaze-based side-channel evaluation platform.

(1) The NVDLA's kernel-mode driver utilizes certain low-level drivers such as Direct Rendering Manager (DRM) for low-level communication with the Linux kernel. In order to successfully insert the kernel module, it is critical to ensure that the correct low-level drivers are available in the system. For a ZYNQ platform, these drivers are enabled by default, but this is not true for a Microblaze system. The correct drivers must be manually enabled while compiling the PetaLinux project. In our design, we used PetaLinux 2019.1 with all the required drivers (DRM, etc.) enabled for NVDLA support.
(2) Microblaze support was not natively provided by NVIDIA. Further, the Linux kernel version is different in our case. As a result, we ported and recompiled the user-mode (consisting of *libnvdla_compiler* and *libnvdla_runtime*) and kernel-mode (*opendla.ko*) drivers. In addition, the provided precompiled libraries such as *libprotobuf.a* had to be recompiled using cross-compilation tools.
(3) Xilinx does not provide PetaLinux support for 64-bit Microblaze, hence we used the 32-bit Microblaze system. However, the drivers provided by NVIDIA access memory using a 64-bit address which conflicts with a 32-bit based Microblaze. To address this issue, we modified the original driver source code to handle 32-bit addresses instead of 64-bit addresses.
(4) Further, the recompiled *libnvdla_compiler.so* file has to be loaded as a shared library file for *libnvdla_runtime*. However, we encountered issues in loading the *libnvdla_compiler.so* file. The same issue does not arise for a ZYNQ based system, and we believe it might be due to some dependency related to Microblaze. Hence, we combined the source code for both *libnvdla_compiler* and *libnvdla_runtime* for compiling into a single libnvdla file, which is then used at runtime.
(5) In addition to porting the source code to the Microblaze system, the setup for NVDLA requires loading the loadable files for inferencing using an SD-card or using SSH/SCP through ethernet. The SASEBO-GIII does not have support for either. Hence, we wrote some firmware and programmed the SPARTAN-6 FPGA available on-board to bypass the UART. This allowed us to use a custom tool to transfer all the files using a serial port connection from the PC to the board.

These changes allowed for the execution of NVDLA loadable files on the SASEBO board. This was necessary for side-channel trace capture.

3.2 Generation of different models

For the target network, we used CNNs, as they are one of the most widely used neural networks. The network is trained offline using the Caffe framework [16] and then compiled into a loadable file using NVDLA compiler. As shown in Fig. 1a, a 4-stage process is used to generate a single loadable file. The loadable file is then used to run inferencing using the NVDLA accelerator on SASEBO-GIII.

For proof of concept, we trained a 4-layer DNN. The base target architecture consists of the following layers: one convolution layer, one activation layer (ReLU), one pooling layer, and a fully connected layer. The input size is 28×28 and the architecture is trained on the MNIST dataset [25]. In this work, our main goal is to highlight the observed differences in power leakage patterns based on different network parameter variations. For this, we trained around 42 different models with variations in different kernel sizes, different number of filters, different filter sizes, etc. for the convolution layer. Similarly, with different strides, different kernel sizes for the pooling layer as well. The complete list of models is provided in Appendix C. One can create many other variations of models with many more layers or parameter variations. But the basic idea for the attack remains the same as discussed in this work.

3.3 Difficulty in full trace capture

Another challenge is to capture the full trace corresponding to a complete inference operation. A simple four-layer CNN requires about 0.5 seconds for inference on NVDLA. In order to capture the full execution trace, one needs to have a high sampling rate, thus requiring a large memory in the oscilloscope. This is necessary as a lower sampling rate results in layers diminishing around the noise level because of low accuracy, which makes it quite challenging to distinguish between the NN layer execution and the background noise. A high-resolution oscilloscope might not be readily available to everyone as it is in our case. So, we used the approach of capturing the trace in multiple smaller chunks. The idea is to run the same inference on repeat and advance in time by setting the correct trigger delay in the oscilloscope and saving the trace before performing the advance. We wrote a small GUI-based application to automate this process. This approach allowed us to capture a long trace and maintain a high sampling rate of around 2GS/s.

3.4 Downsampling algorithm for visualization

Almost all the captured traces consist of more than 400K sample points, in some of the traces there are even millions of sample points. Such a high sampling rate is necessary to observe the small differences in power leakages, especially when multiple kernels/filters are executing in parallel. But, it is difficult to plot a trace without downsampling these many sample points using the commonly available applications. Since our main objective is reverse engineering the structure, it is very important to preserve the overall pattern of the captured traces.

We first evaluated different known approaches such as averaging, decimation, interpolation, etc. But the results obtained after downsampling ended up being lossy and important/expected peaks were either not visible clearly or were hard to distinguish from the noise. To overcome this, we devised a sliding window max-min based downsampling technique (Algorithm 1) and used it in our experiments. The intuition behind this is that we want to preserve maximum and minimum peaks alternatively. For this, we first calculate the maximum and minimum values in the current window (lines 10-15). Then, we alternatively store these values (lines 17-20) while sliding the window with 50% overlap (line 21). This ensures that the leakage pattern is not lost after downsampling.

3.5 Need for signal filtering

For a successful structure recovery using side-channel attacks, it is necessary to capture traces with leakage corresponding to the desired operation. However, since NVDLA is a highly optimized accelerator, multiple operations are executed in parallel and in a pipelined manner. For example, the accumulation starts in parallel with the convolution operation. Similarly, the bias addition also starts in parallel as soon as the updated data is available. Apart from this, data transfer from DRAM also sometimes happens in parallel. Moreover, we have random noise coming from Linux tasks running in the background. So, it is not an ideal setup free of noise. There are many different components that interact with each other, resulting in overall leakage. One possible way to reduce this noise is by using filters in the setup. We experimented with several combinations of custom high-pass, low-pass and band-stop filters as many of the noise components are unknown and cannot be isolated using a single filter.

In our design, NVDLA is running at 50 MHz and the OS is running at 100 MHz. To characterize the noise profile, we started with seven low-pass filters (10, 35, 50, 75, 100, 200, 300 MHz). We sequentially evaluated these filters for suitability in removing high-frequency components. Using visual inspection, we observed that the 75 and 100 MHz filters yield good signal quality for convolution and fully connected layers. But this does not suffice to discriminate the pooling layer in all cases. This is due to the fact that pooling layer performs very low-complexity operations such as averaging, which results in very low power leakage leading to low signal amplitude in the trace. Furthermore, a large amount of noise is present in the lower frequency ranges (5 - 20 MHz), which dominates the overall leakage profile. To overcome this and make the pooling layer signal more

Algorithm 1: Max-Min sliding window downsampling

Input: T // array consisting of captured trace data
Output: S // array consisting of downsampled trace data
Data: *tlen* // denotes the length of captured trace
slen // denotes the length of downsampled trace

```
i := 0
final_samples := slen/2
/* denotes the length of samples considered in a window */
samples_per_block := tlen/final_samples
current_block_pos := 0
while i < (final_samples * 2 - 2) do
    j := 0
    W_max := INT_MIN
    W_min := INT_MAX
    /* find maximum and minimum in the current window */
    while j < samples_per_block do
        current_value := T[current_block_pos + j]
        if current_value > W_max then
            W_max := current_value
        if current_value > W_min then
            W_min := current_value
        j := j + 1
    /* save the datapoints */
    if i % 2 == 0 then
        S[i] := W_max
    else
        S[i] := W_min
    /* slide window with overlapping half points */
    current_block_pos = current_block_pos + (samples_per_block/2)
return S
```

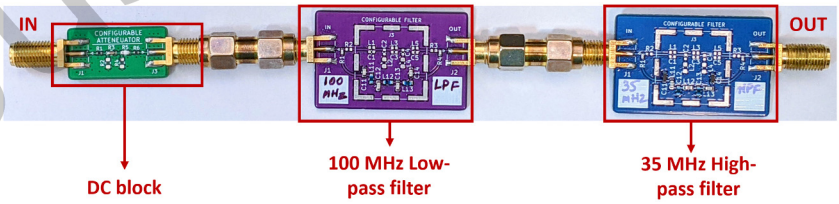


Fig. 2. Filter setup

clear, we tested high-pass filters ranging from 10-50 MHz. It was observed that using a 35 MHz filter resulted in a very clear signal for the pooling layer as well.

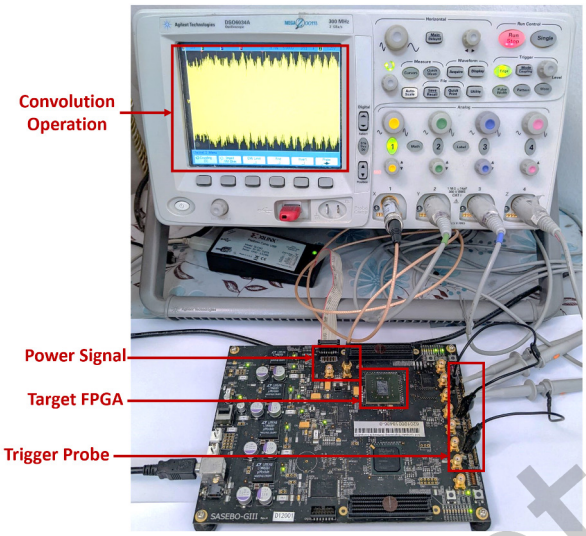


Fig. 3. Overall measurement setup

In general, we observed that using higher order filters provides better signal resolution. We ended up using seventh-order Butterworth filters to ensure low disortion in the passband. Moreover, we observed that the sequence of high-pass and low-pass filters in the setup provided quite close but slightly different results. So, we used the best sequence in terms of signal quality for our experiments.

As a further attempt to block the clock noise from the Microblaze, we applied a 95-105 MHz fifth-order band-stop filter. But, this had a very negligible effect on the signal clarity and thus, we did not use it for our evaluation. This can be attributed to the fact that the NVDLA accelerator is significantly larger than the Microblaze. Finally, using a 100 MHz low-pass filter followed by a 35 MHz high-pass filter provided the best results for the signal as shown in Fig. 2. Hence, we used this combination to capture all the traces in our experiments. In addition, a DC block was used to remove the 1V bias voltage from the FPGA VCCINT signal.

3.6 Final measurement setup

An Agilent DSO6034A oscilloscope was used to capture the traces from the SASEBO board. The measurement setup is shown in Fig. 3. A custom GUI application was developed to control the oscilloscope and the SASEBO board using USB and UART interfaces respectively. As the signal levels are quite low, a custom low-noise, wide-band 20dB amplifier utilizing an Analog Devices HMC8410 MMIC was used to boost the signal levels. The amplifier helps to improve the SNR and overall experimental results.

4 Model Extraction

Here, we describe how a combination of SPA and timing side-channel attacks can be used to extract different parameters of a neural network. The traces were captured at 2GS/s, but for visual perspective we have shown the downsampled traces except for the zoomed versions. The downsampling was performed using Algorithm 1. According to the NVDLA documentation, only ReLU and PReLU are supported for the *nv_small* configuration. However, of these two, only ReLU activation is supported in *nvdla_compiler* [34] for parsing. Hence, we only focus on identifying and differentiating between convolution, pooling, and fully connected layer and their respective

parameters. However, for different activation functions, it is possible that the power trace differs in certain ways, like the amplitude and shape of the peak. But, without actual traces it is hard to speculate in this regard.

4.1 Identifying different number of layers

The first target in structure recovery is to identify the number of layers in the neural network model. The complete model takes about a few seconds to finish execution using NVDLA. Fig. 4 shows the power trace for the full execution of the LeNet model. As can be seen, there are three clearly distinguishable peaks. They correspond to the MAC operations in Convolution or Fully connected layer. Between the peaks, several operations such as Activation layer, Pooling layer, memory transfers, OS scheduling delays happen. But they are not visible clearly in the trace because of the limited sampling rate required to capture the full execution. The trace is captured at only 2MS/s, whereas the rest of the power traces are captured at 2GS/s.

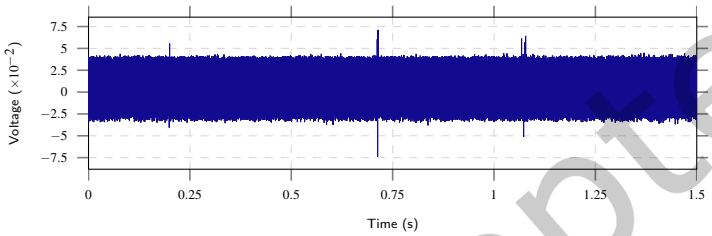


Fig. 4. LeNet full execution power trace

These peaks act as first points of interest for an attacker to investigate to zoom in and capture at high sample rates. Thus, further results in visibility for other operations such as pooling layer as well. The position of the peaks is repeatable in the experiments and can be easily recognized using automated techniques such as template matching or a trained AI model which are then used for profiling phase. Due to the memory limit of

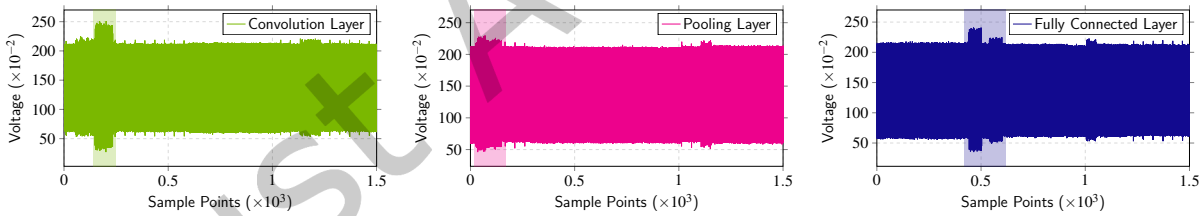
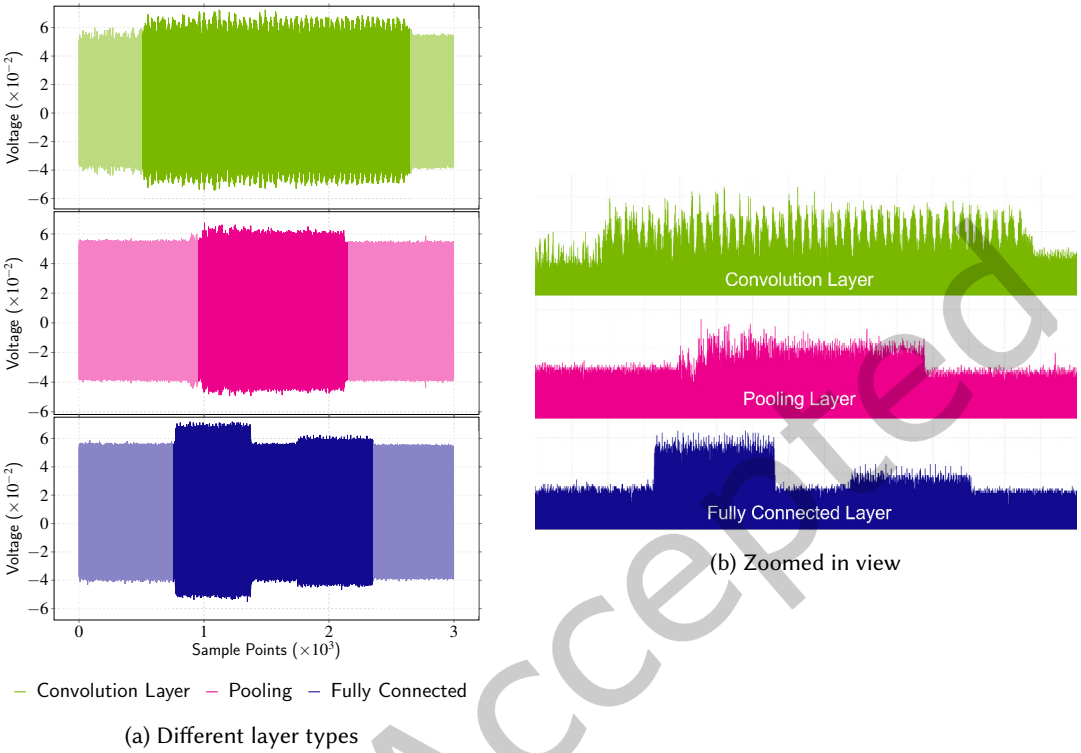


Fig. 5. Full trace execution for different layer types. The highlighted area shows leakages from NVDLA, rest of the leakages are due to the Linux OS running in the background.

the oscilloscope, we captured the trace for this part in multiple chunks as described in Section 3.4. In Fig. 5, we show partial execution windows of about 5ms each, corresponding to different layer types. One can see that in all the figures, there is one significant peak (highlighted in the figure) along with multiple smaller peaks. The higher peaks are the times when the corresponding layer is being executed in the hardware. However, we believe that the smaller peaks either correspond to Linux OS performing some task in the background or data being transferred from BRAM or DRAM for the next layer execution. It is quite evident that the larger peaks corresponding to model operations are distinguishable from the rest of the peaks. But it is important to note that careful inspection of the full model execution may be required to accurately identify the number of layers. This is because the execution time for the corresponding operation is quite short (a few microseconds).

4.2 Distinguishing different layer types



(a) Different layer types

(b) Zoomed in view

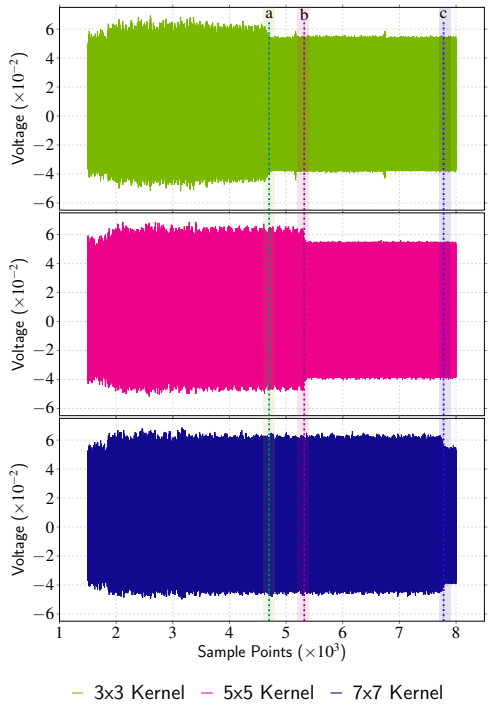
Fig. 6. Power trace showing detailed view for the different layer types. The stark differences due to the different layer operations can be easily seen and distinguished.

Fig. 6a shows the power trace corresponding to the different layers. The regions of interest are highlighted in the figure. The power trace for all the layers is quite different from one another and can be easily recognized. As both convolution and fully connected layer perform the MAC operation, they consume a lot of power, and the leakage is quite high compared to a pooling layer. This is clearly visible from the zoomed-in traces shown in Fig. 6b. Further, due to the nature of implementation of the convolution layer (not all the nodes are connected to one another), we observe more distinct equally spaced peaks in the power trace. However, in case of a fully connected layer, the peaks are always continuous and fine-grained.

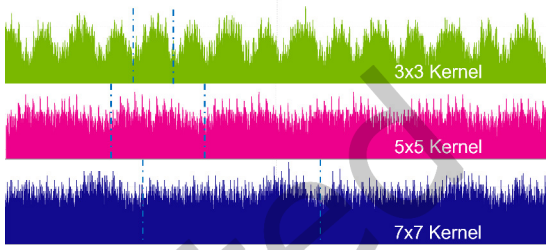
4.3 Reverse engineering of the convolution hyperparameters

4.3.1 ***Kernel size.*** The next target is to identify the kernel size. The most commonly used kernel sizes are 3×3, 5×5 and 7×7. So, in our attack, we focused on only these three kernel sizes. The power trace obtained is shown in Fig. 7a. The points a, b, and c mark the end of convolution operation for different kernel sizes.

One can note that for a 3×3 kernel, the convolution finishes earlier compared to a 5×5 and a 7×7 kernel. This is marked as point a. Similarly, point b marks the end of the convolution operation with a 5×5 kernel and point c denotes the end of convolution with a 7×7 kernel. This shows that there is a significant difference in execution time for different kernel sizes.



(a) Different kernel sizes



(b) Zoomed in view

Fig. 7. Variance in execution time and peak patterns with respect to different kernel sizes. The differences in the peak widths are also significantly different.

The larger kernel size requires more number of stripe operations [35]. As a result, the width of the individual peaks visible in Fig. 7a is different for different kernel sizes. This is demonstrated in Fig. 7b. For better visualization, we have highlighted the width of a single peak for each kernel size. Hence, using timing analysis, one can easily find out the kernel size. One can also note that the highlighted pattern is repeated continuously.

4.3.2 ***Number of output nodes.*** As we used *nv_small* configuration for our attack, the number of possible kernels that can be executed in parallel is 8. So, our aim now is to find out how many kernels are being executed in parallel. As shown in Fig. 8, the power trace corresponding to the number of parallel kernels = 1,2,4 and 8 versus the number of kernels = 3,5,6 and 7 differs significantly. The trace corresponding to kernels that are equal to a power of 2 have finer peaks, whereas in other cases the peaks are wider. Interestingly, the former also requires significantly less time for execution compared to the latter. We believe this is because of how the MAC operations are scheduled inside NVDLA.

Furthermore, in our experiments, we observed that the power leakage is additive as shown in Fig. 9. The figure shows power consumption traces when the number of kernels executed in parallel is 1, 2, 4 and 8 respectively. One can see that with an increasing number of kernels, the trace amplitude increases as well. For better clarity, we also show a zoomed in version of the trace. Using this observation, one can build a power template and use it to recover the number of kernels being executed in parallel. This is also true in the case when number of kernels executed is 3,5,6 and 7 as shown in Fig. 10.

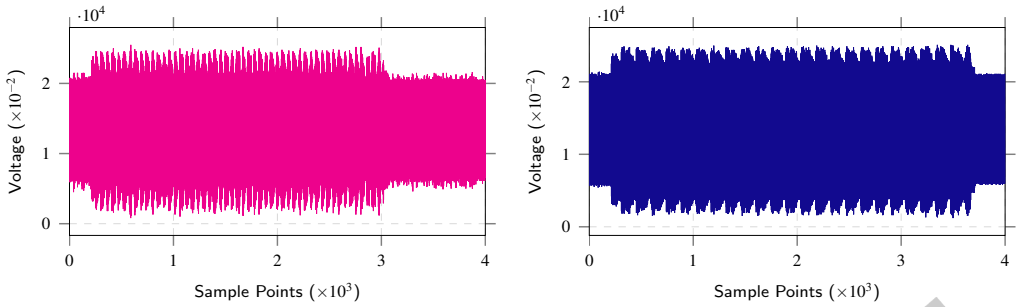


Fig. 8. Number of parallel (3×3) kernels. Side channel leakage pattern when the number of parallel kernels is a power of two versus the others. Leakages for [1,2,4,8] are on the left, while leakages for [3,5,6,7] are on the right.

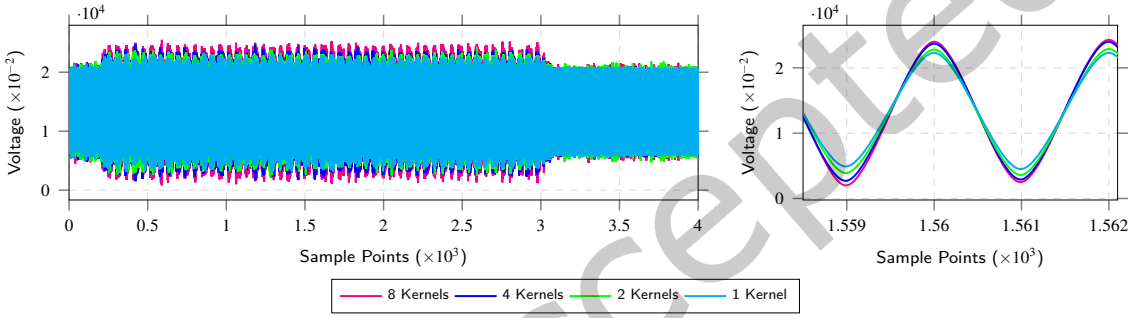


Fig. 9. Number of parallel (3×3) kernels. The amplitude difference is clearly visible in the zoomed in view.

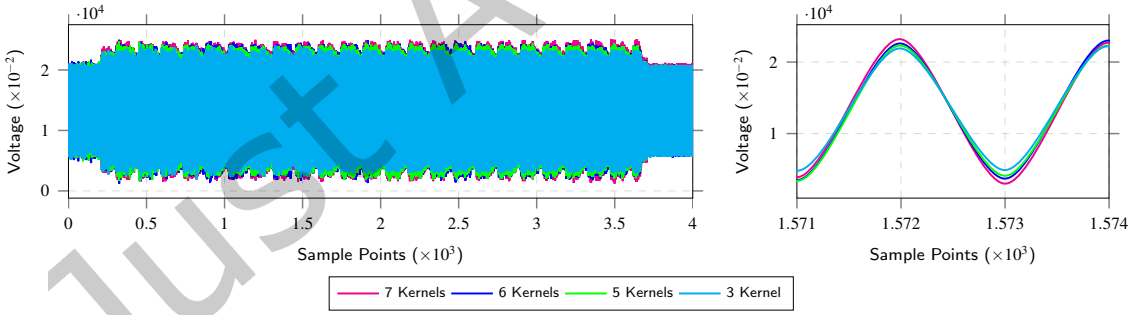


Fig. 10. Number of parallel (3×3) kernels. The amplitude difference is clearly visible in the zoomed in view.

NVDLA small allows the execution of only 8 kernels in parallel. Hence, if the number of kernels in the model is more than 8, then they are executed in batches of 8. For instance, if the model has 20 kernels to be executed, then three batches will be created with 8, 8 and 4 kernels. Individual batches are also visible on the power trace when they are executed one after another. We demonstrate this for different kernels in Fig. 11 where we show the execution of 1 kernel versus 20 kernels. One can see that the time required to finish the execution of one kernel is repeated three times for 20 kernels.

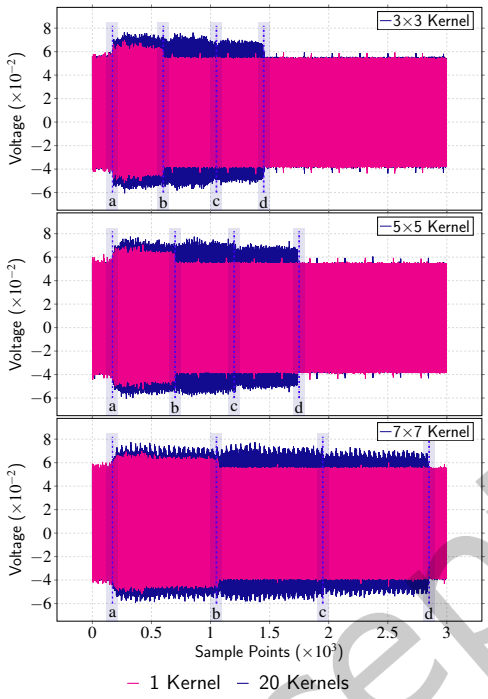


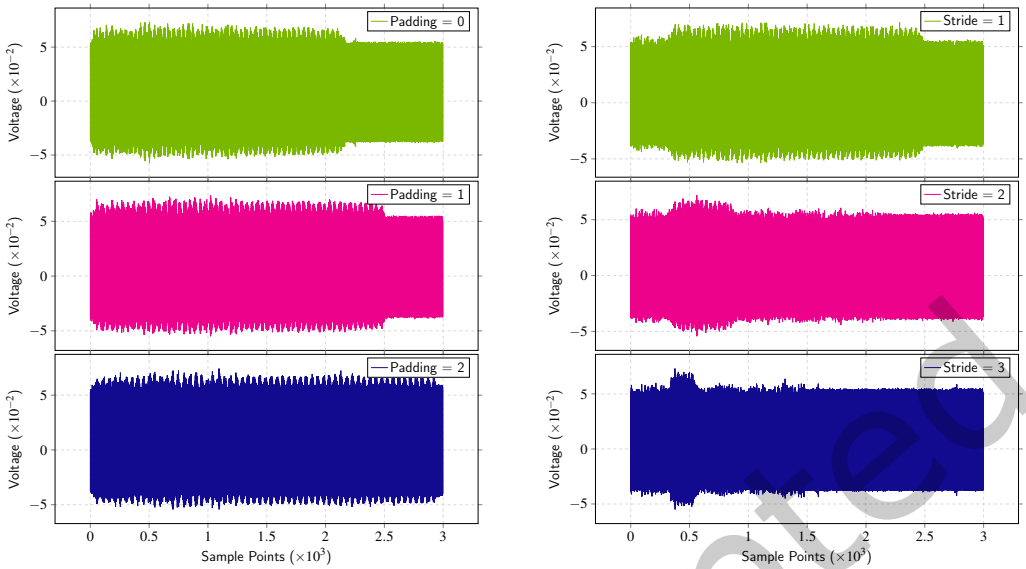
Fig. 11. Effect of batch and kernel size: Increase in either of them results in an overall higher execution time.

The points a-b, b-c, and c-d in the figure mark three batch executions for the current model. As there are three batches, it can be concluded that there are a maximum of 24 kernels in the architecture. In order to obtain the exact number of kernels, the strategy discussed for Fig. 8, 9 and 10 can be utilized to deduce the exact number of kernels for the last batch. One can also observe that the distance between points a-b, b-c and c-d are different for different kernel sizes. This is the same time execution difference that we exploited earlier in identifying the different kernel sizes. Another interesting thing to note is that for region a-b, there is a difference in signal amplitude due to parallel execution of one versus eight kernels.

4.3.3 ***Padding***. Padding means adding empty pixels at the edge of an input. This is a very common technique typically used in neural networks to preserve the boundaries of an input and to prevent shrinking of the input after each convolution layer. The padding applied to an input is constrained by the fact that it should always be less than the kernel size used for that specific layer. For example, if the kernel size is 3×3, then the padding size can only be less than 3. Hence, we show the power trace obtained only for valid padding cases. Fig. 12a shows the difference in padding possible for a 3×3 kernel size. One can note that as the padding size increases, the number of individual peaks also increases. Furthermore, the width of the peak does not change as the kernel size does not change.

Similar behaviour is observed for padding corresponding to a 5×5 kernel. In this case, the possible padding sizes are 0, 1, 2, 3 and 4 and the observed power traces are shown in Fig. 17a in the appendix. An interesting thing to note is that the increase in number of peaks or execution time is consistent with the increase in padding size.

4.3.4 ***Stride***. A common strategy to apply a kernel over an input image is to slide the kernel. The step size used for sliding the kernel is defined by the stride parameter. A stride of more than one is most commonly used for



(a) Padding for 3x3 kernel

(b) Stride for 3x3 kernel

Fig. 12. Power profiles corresponding to varied padding and stride size. Increase in padding size leads to longer execution time, whereas it is opposite for the stride size.

downsampling and for computational efficiency. The stride for a layer is usually constrained by its kernel size and is defined to be less than or equal to the kernel size. For instance, if the kernel size is 3×3, then the valid strides can be 1, 2 and 3. Fig. 12b shows the difference between different strides for a 3×3 kernel. In addition, the stride is applied when sliding from left to right and from top to bottom. Thus, increasing the stride significantly reduces the number of stripe operations and hence the overall time required for the convolution operation. This is also clearly visible from the power trace. When stride=3, the convolution execution period is significantly shorter compared to stride=1.

4.4 Reverse engineering fully connected layer

The strategy discussed in Section 4.3.2 to extract the number of output nodes in a convolution layer also applies to the fully connected layer as well. This is because both uses the same MAC engines for execution and as a result 8 kernels can be executed in parallel. Fig. 13 shows the power leakage corresponding to a fully connected layer with 500 output nodes ($62 \times 8 + 4$). One can see 62 long peaks each performing computations for 8 output nodes and the smaller peak at the end corresponds to the remaining 4 output nodes.

4.5 Reverse engineering pooling layer

Apart from recovering different parameters for the convolution layer, we also performed some experiments to observe leakage patterns for a pooling layer to identify its kernel size and stride. Compared to a convolution layer, there is no stripe operation in a pooling layer. Instead, the throughput of a pooling layer for *nv_small* architecture is 1. This means that if one uses a 3×3 kernel with Average Pooling, then the complete kernel is applied in one clock cycle to generate the output. Hence, the observed leakage pattern is not the same as in a convolution layer.

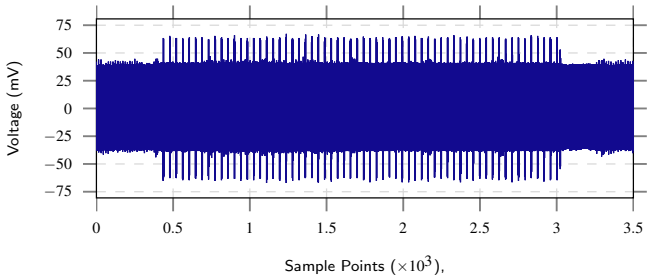


Fig. 13. Power trace showing leakage for computation of 500 nodes for a fully connected layer.

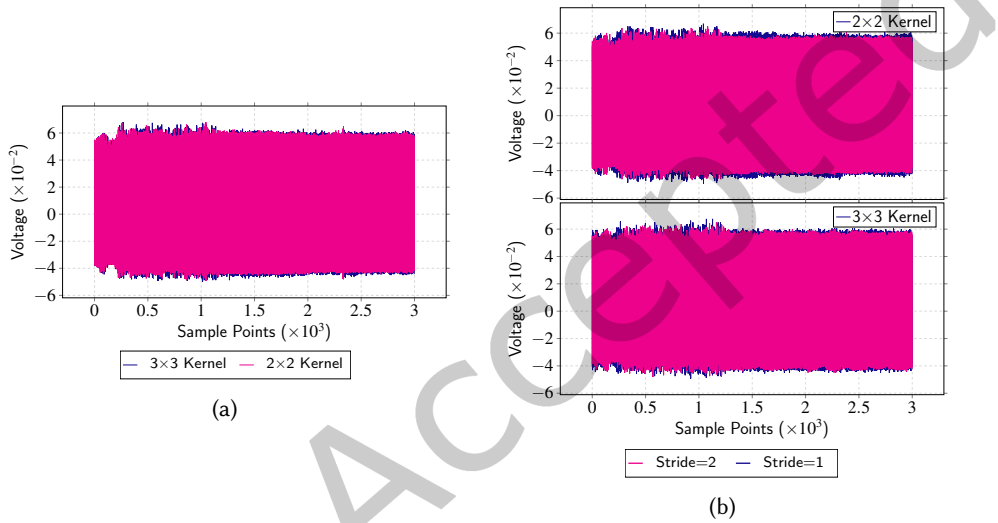


Fig. 14. Power trace for Pooling layer kernel size (on the left) and differences in stride size (on the right).

4.5.1 ***Kernel size***. The traces obtained for two different kernel sizes 2×2 and 3×3 are shown in Fig. 14a. It is quite evident that the power consumption required when the kernel size is 3×3 is consistently greater compared to when it is 2×2. This might be because more pixels are being processed in a single clock cycle in the case of a 3×3 kernel compared to a 2×2 kernel.

4.5.2 ***Stride***. Fig. 14b shows the traces for different strides. It is interesting to note that the power consumption required when stride=1 is higher compared to when stride=2. This is true for both a 2×2 and a 3×3 kernel. We believe this because when stride=2, the amount of processing is less compared to stride=1.

4.6 AI-assisted automated attack flow

Fig. 15 presents the complete flow of our AI-assisted attack. In step ❶, we develop and train multiple NN models using the MNIST dataset. Then, we use trained models for inferencing in SASEBO-GIII (step ❷) to obtain side-channel power profiles (step ❸) as shown in Section 4. One should note that the generated profiles and models are for different layer types and their parameters such as different kernel sizes, stride sizes etc. and not the complete model. Hence, the same power profiles can be used for widely varying CNN models. These profiles are used as

datasets to train the CNN attack models in step ❹. The steps ❶-❹ constitute the initial profiling phase of the attack.

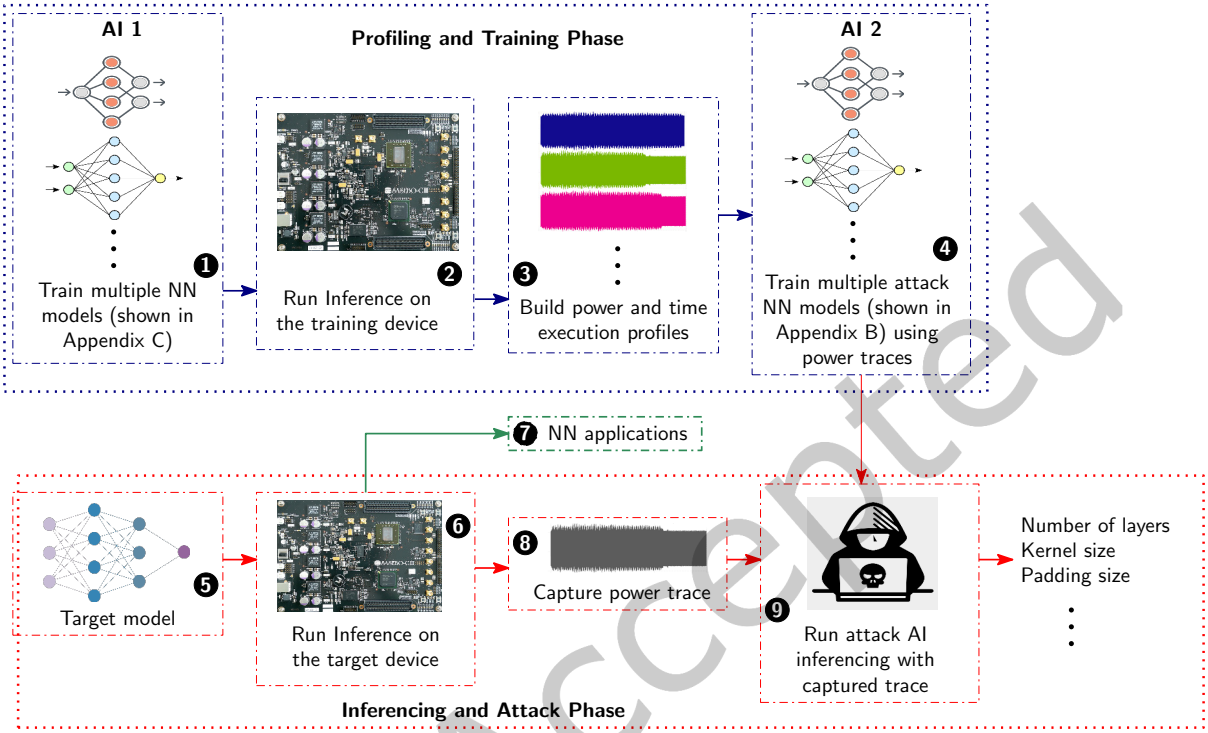


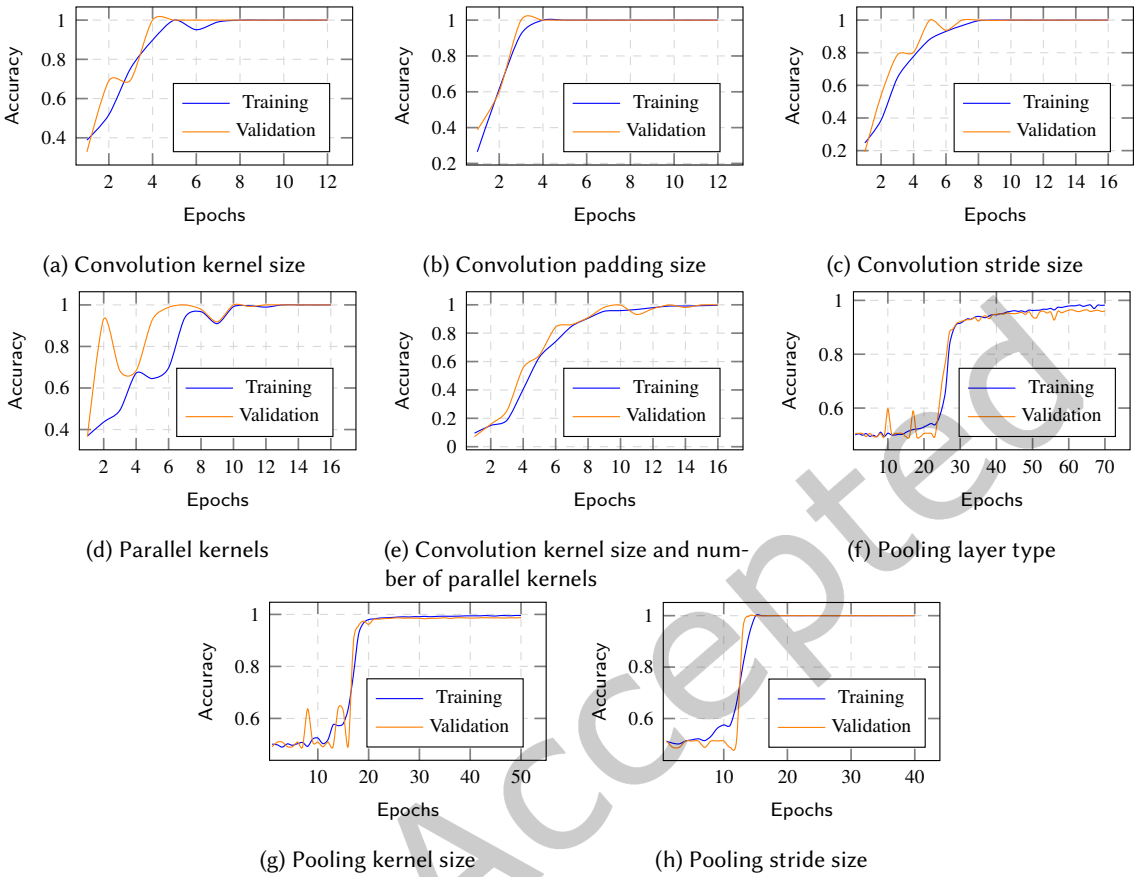
Fig. 15. Overview of the AI-assisted attack flow.

In steps ❺ and ❻, the target victim model is used for the inference of the actual application and the final result is provided in step ❼. When the hardware platform is running the actual inference, it is leaking data which the attacker is capturing in step ❽. The final part of the attack (step ❾) is to utilize the trained attack models (from step ❹) for inferencing with the captured power trace to recover the respective neural network parameters.

5 Results and Evaluation

In this section, we demonstrate the attack and present results for the same. For this, we trained eight attack models to reverse engineer different parameters of the convolution and pooling layer. The attack models were trained using Keras and Tensorflow back-end. The AI is trained on upto 30,000 sample datapoints downsampled from more than 2 million datapoints. As a result, AI has a much richer dataset on which to learn and train. The details of the respective model architecture are provided in Appendix B. As mentioned in section 4.6, the datasets for all the attack models were generated using a subset of models from Appendix C. Further, we used our trained models for inferencing the structure parameters of the LeNet network.

Fig. 16 shows the accuracies of different trained models. For all experiments, we used 80% of the dataset as training data and 20% as validation data. As the differences between the parameters were quite visible in the power traces, the AI-based trained models demonstrate high accuracy. As shown in section 4.3.1, the difference in kernel sizes is clearly visible, and as a result the model is trained faster with very few traces and epochs. Whereas,



(a) Convolution kernel size (b) Convolution padding size (c) Convolution stride size

(d) Parallel kernels (e) Convolution kernel size and number of parallel kernels (f) Pooling layer type

(g) Pooling kernel size (h) Pooling stride size

Fig. 16. Training and validation accuracy for different attack models

the traces for the pooling layer type were visually indistinguishable. This can be attributed to the fact that the throughput of the pooling layer is one. Hence, the time execution will remain the same no matter what type of pooling is used, and the power consumption is also quite similar. But, training the AI with more traces and epochs we were able to recover the pooling layer type with a very good accuracy as well.

Apart from analyzing the traces to recover a single parameter such as padding size, stride size, etc., we also attempted to recover two parameters simultaneously. Fig. 16e shows results for recovering kernel size and number of parallel kernels that execute in a convolution layer. It can be seen that the AI can recover both of these parameters in a single run with high accuracy. Such optimizations can lead to significant speedups in structural reverse engineering.

Table 2 presents the test accuracies for different trained models. For our evaluation, we captured 100 traces from LeNet using random inputs for inferencing corresponding to the first convolution and pooling layers. One can see that the test accuracies for our model as well as LeNet are quite high. This validates the effectiveness of our trained attack models for reverse engineering the structure of the victim model and its parameters. In some cases such as in recovery of pooling layer type, the accuracy was around 95% resulting in misprediction of the network parameter. But we observed that the incorrect predictions are not very far from the correct ones.

So, the recovered model structure even in the presence of these errors is close to the original and is expected to give similar accuracy. Additionally, the attacker knows the error rates of the attack model and as a result can fine-tune the specific model parameters which are more error-prone (for example AVG or MAX pooling in our case) if the recovered model does not perform optimally. The parameters that are recovered with high confidence may not be changed or tweaked.

Table 2. Summary of the test accuracies. Our models consist of power traces captured during execution of a subset of models from Table 10, Appendix C.

Ref. as	Layer	Architectural	Our models			LeNet	
		Parameters	Model #	Test Set	Acc.	Test Set	Acc.
AM1	Conv	Kernel size	9,11,13	1500	100%	100	100%
AM2	Conv	Padding size	11, 36-39	750	100%	100	99%
AM3	Conv	Stride size	11, 32-35	1000	99.6%	100	98%
AM4	Conv, FC	Parallel kernels	10,11,40	1500	100%	100	100%
AM5	Conv	Kernel size and Parallel kernel	1,2,4, 8-12, 40-43	960	99.9%	100	97%
AM6	Pool	Type (AVG or MAX)	14,28	2000	96.7%	100	95%
AM7	Pool	Kernel size	14,30	1000	99.0%	100	99%
AM8	Pool	Stride size	1,29	2000	100%	100	99%

Table 3 presents a summary of the attack results. One can see that nearly all the architectural parameters can be recovered using our attack. We did not attempt to recover the padding size for the pooling layer. We believe similar to kernel and stride size, padding size should be recoverable too as in the case of a convolution layer.

6 Further Discussions and Future Directions

- **Limitations.** Even though this work shows that it is possible to recover the architecture of NN models with good accuracy, the attacker still needs a good data set for training. This is because the neural network weights were not targeted. Recovery of inputs and weights require a different form of attack like Differential Power Analysis and as a result the trace capture setup and processing steps are significantly different compared to this work. This is left as a possible future extension.
- **Applicability towards other real-world applications.** Real-world applications involving huge datasets typically require deep and complex models such as AlexNet, VGGNet, ResNet, etc. Even though the models are quite complex, the building blocks (convolution, pooling, etc.) remain the same. Since the side-channel leakage is clearly visible in the power traces, and most of the network parameters are recoverable, the attack techniques demonstrated in this work can be directly applied to recover other larger models as well. Such architecture recovery would allow a motivated attacker to obtain and utilize models from commercially deployed products, significantly reducing the time and effort required to train a custom model from scratch.
- **Possible side-channel countermeasures.** Side-channel countermeasures for secure ML are an open topic. Unless we have a deep integration of cryptographic techniques in NN operation, it is not possible to directly lift the SCA countermeasures from cryptography to ML. Typical side-channel countermeasures like masking are used to protect secrets such as inputs, weights, etc. [8–10]. The masking countermeasure targets the mathematical structure and uses randomization to protect specific values such as weights and inputs. But it does not protect the overall operation and information such as number of layers, type of layers, etc. Further, typical cryptographic countermeasures such as shuffling, reordering of operations, etc. cannot be applied to protect NN reverse engineering. This is because the sequence of operations remains

Table 3. Summary of the results. Recovered parameters are denoted by ✓ and the parameters which we did not attempt to recover are denoted by – .

Architectural Parameters	Result
Number of layers	✓
Type of layers	✓
Sequence of layers	✓
Convolution layer	
Number of parallel executing kernels	✓
Total number of kernels	✓
Kernel size	✓
Stride size	✓
Padding size	✓
Pooling layer	
AVG or MAX	✓
Kernel size	✓
Stride size	✓
Padding size	–
Inputs	–
Weights	–

the same and is still visible in the power traces. In addition, there has been some progress in protecting the structure using obfuscation techniques [26, 38] such as layer widening, layer branching, scheduling, etc. However, a detailed evaluation of such techniques on hardware platforms with power traces is yet to be explored.

7 Conclusion

In this paper, we present an in-depth evaluation of a widely deployed commercial NN accelerator from NVIDIA. To the best of our knowledge, this is the first such work in this direction. We first show that the network parameters and hyperparameters such as padding, stride, and kernel size are distinguishable using SPA and timing side-channel attack. Then, we utilized these power traces to train attack AI models achieving very high accuracy. These trained models were then used to recover network parameters from LeNet execution on NVDLA with more than 95% accuracy. We also provide details about how to overcome the challenges faced due to a highly pipelined and parallel hardware architecture and capture traces with good SNR.

Availability

We will provide the relevant source code and dataset for the tools developed at https://github.com/nainag after publication.

References

[1] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. CSI NN: Reverse Engineering of Neural Network Architectures through Electromagnetic Side Channel. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 515–532.

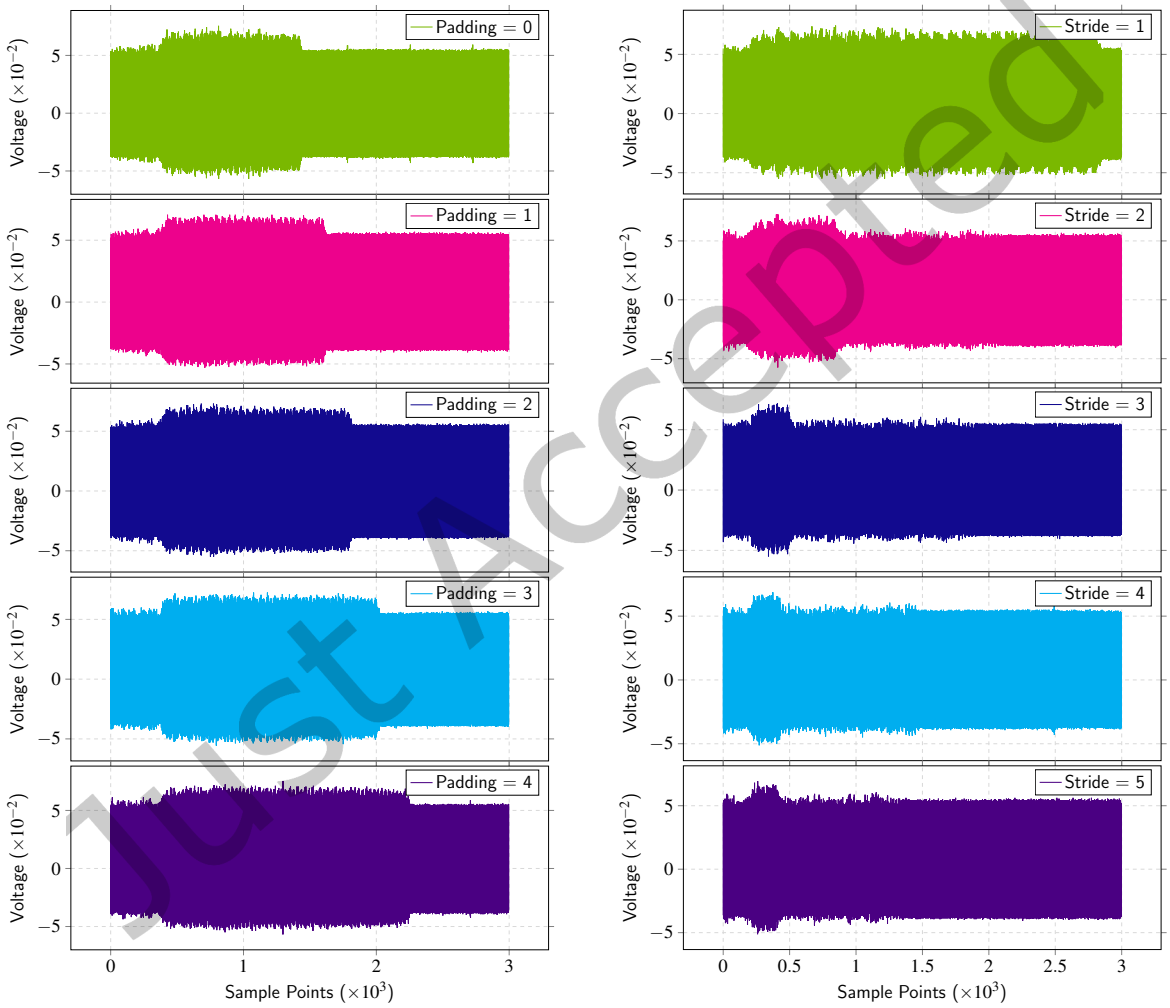
[2] Vincent Carlier, Hervé Chabanne, Emmanuelle Dottax, and Hervé Pelletier. 2004. Electromagnetic side channels of an FPGA implementation of AES. In *CRYPTOLOGY EPRINT ARCHIVE, REPORT 2004/145*. Citeseer.
[3] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. 2021. Side channel attacks for architecture extraction of neural networks. *CAAI Transactions on Intelligence Technology* 6, 1 (2021), 3–16.
[4] Łukasz Chmielewski and Léo Weissbart. 2021. On reverse engineering neural network implementation on gpu. In *Applied Cryptography and Network Security Workshops: ACNS 2021 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, and SiMLA, Kamakura, Japan, June 21–24, 2021, Proceedings*. Springer, 96–113.
[5] Elke De Mulder, Pieter Buysschaert, SB Ors, Peter Delmotte, Bart Preneel, Guy Vandenbosch, and Ingrid Verbauwhede. 2005. Electromagnetic analysis attack on an FPGA implementation of an elliptic curve cryptosystem. In *EUROCON 2005-The International Conference on" Computer as a Tool"*, Vol. 2. IEEE, 1879–1882.
[6] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. 1998. A practical implementation of the timing attack. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 167–182.
[7] Xiaoyi Duan, Qi Cui, Sixiang Wang, Huawei Fang, and Gaojian She. 2016. Differential power analysis attack and efficient countermeasures on PRESENT. In *2016 8th IEEE International Conference on Communication Software and Networks (ICCSN)*. IEEE, 8–12.
[8] Anuj Dubey, Afzal Ahmad, Muhammad Adeel Pasha, Rosario Cammarota, and Aydin Aysu. 2022. ModuloNET: Neural Networks Meet Modular Arithmetic for Efficient Hardware Masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), 506–556.
[9] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. 2020. Bomanet: Boolean masking of an entire neural network. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.
[10] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. 2020. Maskednet: The first hardware inference engine aiming power side-channel protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 197–208.
[11] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2017. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1857–1874.
[12] Peter Horvath, Lukasz Chmielewski, Leo Weissbart, Lejla Batina, and Yuval Yarom. 2023. BarraCUDA: Bringing Electromagnetic Side Channel Into Play to Steal the Weights of Neural Networks from NVIDIA GPUs. *arXiv preprint arXiv:2312.07783* (2023).
[13] Xiaolu Hou, Jakub Breier, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. 2021. Physical security of deep learning on edge devices: Comprehensive evaluation of fault injection attack vectors. *Microelectronics Reliability* 120 (2021), 114116. doi:10.1016/j.microrel.2021.114116
[14] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. 2018. Reverse Engineering Convolutional Neural Networks through Side-Channel Information Leaks. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. doi:10.1145/3195970.3196105
[15] Intel. [n. d.]. Intel Movidius Neural Compute Stick. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-movidius-neural-compute-stick.html.
[16] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.
[17] Raphaël Joud, Pierre-Alain Moëllic, Simon Pontié, and Jean-Baptiste Rigaud. 2023. Like an open book? Read neural network architecture with simple power analysis on 32-bit microcontrollers. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 256–276.
[18] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
[19] Mahdi Khosravy, Kazuaki Nakamura, Yuki Hirose, Naoko Nitta, and Noboru Babaguchi. 2021. Model inversion attack: analysis under gray-box scenario on deep learning based face recognition system. *KSII Transactions on Internet and Information Systems (TIIS)* 15, 3 (2021), 1100–1118.
[20] Mahdi Khosravy, Kazuaki Nakamura, Yuki Hirose, Naoko Nitta, and Noboru Babaguchi. 2022. Model inversion attack by integration of deep generative models: Privacy-sensitive face generation from a face recognition system. *IEEE Transactions on Information Forensics and Security* 17 (2022), 357–372.
[21] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Annual International Cryptology Conference*. Springer, 388–397.
[22] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.
[23] Francois Koeune, Jean-Jacques Quisquater, and Jean-Jacques Quisquater. 1999. A timing attack against Rijndael. (1999).

[24] Ashley Kurian, Anuj Dubey, Ferhat Yaman, and Aydin Aysu. 2025. TPUXtract: An Exhaustive Hyperparameter Extraction Framework. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 1 (2025), 78–103.
[25] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).
[26] Jingtao Li, Zhezhi He, Adnan Siraj Rakin, Deliang Fan, and Chaitali Chakrabarti. 2021. NeurObfuscator: A Full-stack Obfuscation Tool to Mitigate Neural Architecture Stealing. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 248–258.
[27] Saurav Maji, Utsav Banerjee, and Anantha P. Chandrakasan. 2021. Leaky Nets: Recovering Embedded Neural Network Models and Inputs Through Simple Power and Timing Side-Channels—Attacks and Defenses. *IEEE Internet of Things Journal* 8, 15 (2021), 12079–12092. doi:10.1109/JIOT.2021.3061314
[28] Stefan Mangard. 2002. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *International Conference on Information Security and Cryptology*. Springer, 343–358.
[29] Adam Matthews. 2006. Low cost attacks on smart cards: the electromagnetic sidechannel. *Next Generation Security Software, Sept* (2006).
[30] Rita Mayer-Sommer. 2000. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 78–92.
[31] Maria Mendez Real and Ruben Salvador. 2021. Physical Side-Channel Attacks on Embedded Neural Networks: A Survey. *Applied Sciences* 11, 15 (2021). doi:10.3390/app11156790
[32] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. 1999. Investigations of Power Analysis Attacks on Smartcards. *Smartcard* 99 (1999), 151–161.
[33] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. 2021. Power Side-Channel Attacks on BNN Accelerators in Remote FPGAs. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11, 2 (2021), 357–370. doi:10.1109/JETCAS.2021.3074608
[34] NVDLA. 2018. NVDLA PReLU Issue. https://github.com/nvdla/sw/issues/16#issuecomment-384517936,https://github.com/nvdla/sw/issues/32.
[35] NVIDIA. [n. d.]. NVIDIA Deep Learning Accelerator. http://nvdla.org/hw/v1/ias/unit_description.html.
[36] NVIDIA. 2018. NVIDIA Deep Learning Accelerator. http://nvdla.org/primer.html.
[37] Daryna Oliynyk, Rudolf Mayer, and Andreas Rauber. 2023. I know what you trained last summer: A survey on stealing machine learning models and defences. *Comput. Surveys* 55, 14s (2023), 1–41.
[38] Yidan Sun, Siew-Kei Lam, Guiyuan Jiang, and Peilan He. 2024. Streamlining DNN Obfuscation to Defend Against Model Stealing Attacks. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.
[39] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. 2018. I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 393–406. doi:10.1145/3274694.3274696
[40] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Arindam Basu, and Shivam Bhasin. 2021. DeepFreeze: Cold Boot Attacks and High Fidelity Model Recovery on Commercial EdgeML Device. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.
[41] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Arindam Basu, and Shivam Bhasin. 2021. WaC: First Results on Practical Side-Channel Attacks on Commercial Machine Learning Accelerator. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*. 111–114.
[42] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Shivam Bhasin, and Arindam Basu. 2021. Time to Leak: Cross-Device Timing Attack On Edge Deep Learning Accelerator. In *2021 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE, 1–4.
[43] Qian Xu, Md Tanvir Arafin, and Gang Qu. 2021. Security of neural networks from hardware perspective: A survey and beyond. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 449–454.
[44] Xiaobei Yan, Xiaoxuan Lou, Guowen Xu, Han Qiu, Shangwei Guo, Chip Hong Chang, and Tianwei Zhang. 2023. MERCURY: An Automated Remote Side-channel Attack to Nvidia Deep Learning Accelerator. In *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 188–197.
[45] Kota Yoshida, Takaya Kubota, Shunsuke Okura, Mitsuru Shiozaki, and Takeshi Fujino. 2020. Model reverse-engineering attack using correlation power analysis against systolic array based neural network accelerator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.
[46] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. 2020. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 209–218.
[47] Santiago Zanella-Beguelin, Shruti Tople, Andrew Paverd, and Boris Köpf. 2021. Grey-box extraction of natural language models. In *International Conference on Machine Learning*. PMLR, 12278–12286.
[48] Yicheng Zhang, Rozhin Yasaei, Hao Chen, Zhou Li, and Mohammad Abdullah Al Faruque. 2021. Stealing neural network structure through remote FPGA side-channel analysis. *IEEE Transactions on Information Forensics and Security* 16 (2021), 4377–4388.

A Additional traces for 5×5 Kernel executions.

Fig. 17 shows power traces corresponding to changes in padding and stride sizes when a 5x5 kernel is used. It can be seen that the traces look similar to the 3x3 kernels and similarly an increase in padding size leads to longer execution time, whereas it is opposite for the stride size. This pattern is also repeated for large kernel sizes.

As discussed in section 4.3.2, NVDLA executes kernels in batches. followed by data transfers. This pattern of kernel execution and data transfer can be easily seen in Fig. 18. The figure shows the execution of 50 kernels. As 8 kernels can be executed simultaneously, we see six prominent batches corresponding to 48 kernel executions. This is followed by the remaining two kernels with smaller peaks.



(a) Padding for 5x5 kernel

(b) Stride for 5x5 kernel

Fig. 17. Power profiles corresponding to varied padding and stride sizes for a 5x5 kernel.

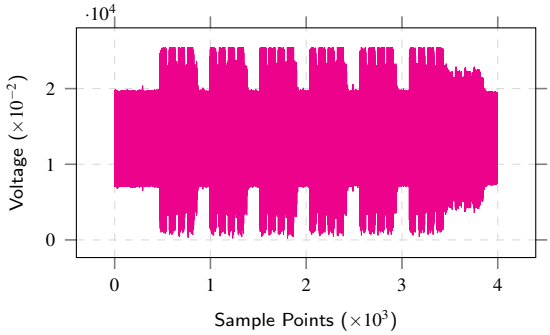


Fig. 18. Power trace for the execution of 50 (5x5) kernels

B Attack Models Architecture and Configuration

Table 4. Model architecture for Attack Model 1 (AM1)

Layer Type	Size	Channels	Filter Size	Stride	Activation
Input Power Trace	200 x 100	1	-	-	-
Conv	200 x 100	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	100 x 50	32	2x2	2x2	-
Conv	100 x 50	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	50 x 25	64	2x2	2x2	-
FC	128	-	-	-	LReLU (slope = 0.1)
FC	3	-	-	-	Softmax

Table 5. Model architecture for Attack Model 2 and 3 (AM2 and AM3)

Layer Type	Size	Channels	Filter Size	Stride	Activation
Input Power Trace	200 x 99	1	-	-	-
Conv	200 x 99	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	100 x 50	32	2x2	2x2	-
Conv	100 x 50	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	50 x 25	64	2x2	2x2	-
Conv	50 x 25	128	3x3	1x1	LReLU (slope = 0.1)
MaxPool	25 x 13	128	2x2	2x2	-
FC	128	-	-	-	LReLU (slope = 0.1)
FC	5	-	-	-	Softmax

C List of Models

Each model in Table 10 is preceded by an input layer and the last layer is a fully connected layer with 10 output nodes corresponding to 10 digits in the MNIST dataset.

Table 6. Model architecture for Attack Model 4 (AM4)

Layer Type	Size	Channels	Filter Size	Stride	Activation
Input Power Trace	200 x 100	1	-	-	-
Conv	200 x 100	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	100 x 50	32	2x2	2x2	-
Conv	100 x 50	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	50 x 25	64	2x2	2x2	-
Conv	50 x 25	128	3x3	1x1	LReLU (slope = 0.1)
MaxPool	25 x 13	128	2x2	2x2	-
FC	-	128	-	-	LReLU (slope = 0.1)
FC	-	4	-	-	Softmax

Table 7. Model architecture for Attack Model 5 (AM5)

Layer Type	Size	Channels	Filter Size	Stride	Activation
Input Power Trace	200 x 100	1	-	-	-
Conv	200 x 100	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	100 x 50	32	2x2	2x2	-
Conv	100 x 50	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	50 x 25	64	2x2	2x2	-
Conv	50 x 25	128	3x3	1x1	LReLU (slope = 0.1)
MaxPool	25 x 13	128	2x2	2x2	-
FC	128	-	-	-	LReLU (slope = 0.1)
FC	12	-	-	-	Softmax

Table 8. Model architecture for Attack Model 6, 7 and 8 (AM6, AM7 and AM8)

Layer Type	Size	Channels	Filter Size	Stride	Activation
Input Power Trace	290 x 100	1	-	-	-
Conv	290 x 100	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	145 x 50	32	2x2	2x2	-
Conv	145 x 50	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	73 x 25	64	2x2	2x2	-
Conv	73 x 25	128	3x3	1x1	LReLU (slope = 0.1)
MaxPool	37 x 13	128	2x2	2x2	-
FC	128	-	-	-	LReLU (slope = 0.1)
FC	2	-	-	-	Softmax

Table 9. Summary of the training configuration.

Attack Model #	Training set size	Training configuration
AM1	3000	epochs=12, batch=64, optimizer=SGD, momentum=0.7, init lr= 0.005
AM2	4000	epochs=12, batch=64, optimizer=SGD, momentum=0.6, init lr=0.005
AM3	4000	epochs=16, batch=64, optimizer=SGD, momentum=0.6, init lr=0.005
AM4	6000	epochs=16, batch=64, optimizer=SGD, momentum=0.6, init lr= 0.005
AM5	6000	epochs=16, batch=64, optimizer=SGD, momentum=0.6, init lr=0.005
AM6	10000	epochs=70, batch=32, optimizer=SGD, momentum=0.9, init lr=0.009
AM7	9000	epochs=50, batch=32, optimizer=SGD, momentum=0.9, init lr=0.008
AM8	10000	epochs=40, batch=32, optimizer=SGD, momentum=0.6, init lr=0.007

Table 10. Trained models architecture for profiling and generating datasets

Model #	Operation	Layer Name	No. of Filters	Filter size	Stride size	Padding size
1	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
2	Convolution	Convolution + ReLU	2	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
3	Convolution	Convolution + ReLU	3	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
4	Convolution	Convolution + ReLU	4	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
5	Convolution	Convolution + ReLU	5	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
6	Convolution	Convolution + ReLU	6	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
7	Convolution	Convolution + ReLU	7	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
8	Convolution	Convolution + ReLU	8	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
9	Convolution	Convolution + ReLU	20	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
10	Convolution	Convolution + ReLU	1	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
11	Convolution	Convolution + ReLU	20	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
12	Convolution	Convolution + ReLU	1	7×7	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
13	Convolution	Convolution + ReLU	20	7×7	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
14	Convolution	Convolution + ReLU	4	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
15	Convolution	Convolution + ReLU	4	3×3	2×2	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
16	Convolution	Convolution + ReLU	4	3×3	3×3	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
17	Convolution	Convolution + ReLU	4	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
18	Convolution	Convolution + ReLU	4	5×5	2×2	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
19	Convolution	Convolution + ReLU	4	5×5	3×3	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
20	Convolution	Convolution + ReLU	4	5×5	4×4	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
21	Convolution	Convolution + ReLU	4	5×5	5×5	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
22	Convolution	Convolution + ReLU	4	3×3	1×1	1×1
	Pooling	Max pooling	1	2×2	1×1	0×0
23	Convolution	Convolution + ReLU	4	3×3	1×1	2×2
	Pooling	Max pooling	1	2×2	1×1	0×0
24	Convolution	Convolution + ReLU	4	5×5	1×1	1×1
	Pooling	Max pooling	1	2×2	1×1	0×0
25	Convolution	Convolution + ReLU	4	5×5	1×1	2×2
	Pooling	Max pooling	1	2×2	1×1	0×0

26	Convolution	Convolution + ReLU	4	5×5	1×1	3×3
	Pooling	Max pooling	1	2×2	1×1	0×0
27	Convolution	Convolution + ReLU	4	5×5	1×1	4×4
	Pooling	Max pooling	1	2×2	1×1	0×0
28	Convolution	Convolution + ReLU	4	3×3	1×1	0×0
	Pooling	Ave pooling	1	2×2	1×1	0×0
29	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
30	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	3×3	1×1	0×0
31	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	3×3	2×2	0×0
32	Convolution	Convolution + ReLU	20	5×5	2×2	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
33	Convolution	Convolution + ReLU	20	5×5	3×3	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
34	Convolution	Convolution + ReLU	20	5×5	4×4	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
35	Convolution	Convolution + ReLU	20	5×5	5×5	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
36	Convolution	Convolution + ReLU	20	5×5	1×1	1×1
	Pooling	Max pooling	1	2×2	2×2	0×0
37	Convolution	Convolution + ReLU	20	5×5	1×1	2×2
	Pooling	Max pooling	1	2×2	2×2	0×0
38	Convolution	Convolution + ReLU	20	5×5	1×1	3×3
	Pooling	Max pooling	1	2×2	2×2	0×0
39	Convolution	Convolution + ReLU	20	5×5	1×1	4×4
	Pooling	Max pooling	1	2×2	2×2	0×0
40	Convolution	Convolution + ReLU	3	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
41	Convolution	Convolution + ReLU	8	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
42	Convolution	Convolution + ReLU	2	7×7	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0

Received 4 March 2024; revised 20 January 2025; accepted 25 March 2025