AI Attacks AI: Recovering Neural Network architecture from NVDLA using AI-assisted Side Channel Attack

Naina Gupta
Nanyang Technological University
Singapore
naina003@e.ntu.edu.sg

Arpan Jati
Nanyang Technological University
Singapore
arpan.jati@ntu.edu.sg

Anupam Chattopadhyay
Nanyang Technological University
Singapore
anupam@ntu.edu.sg

***Abstract*—During the last decade, there has been a stunning progress in the domain of AI with adoption in both safety-critical and security-critical applications. A key requirement for this is highly trained Machine Learning (ML) models, which are valuable Intellectual Property (IP) of the respective organizations. Naturally, these models have become targets for model recovery attacks through side-channel leakage. However, majority of the attacks reported in literature are either on simple embedded devices or assume a custom Vivado HLS based FPGA accelerator.**

On the other hand, for commercial neural network accelerators, such as Google TPU, Intel Compute Stick and NVDLA, there are relatively fewer successful attacks. Focussing on that direction, 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 use power and timing side-channel leakage information from Convolutional Neural Network (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. Our solution is fully automated, and portable to other hardware neural networks, thus presenting a greater threat towards IP protection.

***Index Terms*—AI, ML, IP Stealing, Side-Channel, NVDLA**

1. Introduction

Growing adoption of AI in various application segments have 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 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 classified broadly in terms of the attack objectives. First, when the attacker intends to recover the input data. This is demonstrated in [30] and [27] 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 [11]. Third, also the focus of the current work, is when an attacker intends to 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 works in recent times, there remains 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 case of microcontrollers. These settings considerably simplifies the problem at hand and is not always representative of a practical setup. One of the key requirement for a successful side-channel attack is the precise target and noise-free traces. Both pipelining and parallel execution, common optimization strategies used for FPGAs make it quite difficult for side-channel attacks. This is also noted in a recent paper along this line of research [21], stating that, parallel execution of multiple NN operations on an FPGA/ASIC platform makes the side channel attack considerably harder. This is one of the reason the current literature lacks the analysis and possible attacks on hardware neural network accelerators.

Further, most of the current literature focused on BNN architectures for FPGA platforms such as in [9], [32], [35]. To the best of our knowledge, no prior work has yet investigated CNN architectures on hardware neural network accelerators using the power traces. Furthermore, the analysis and evaluation of 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 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. [12] 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. Later, Batina et. al. [1] explored recovery of neural network structure and weights in a grey box setting targeting micro-controllers using timing and Electromagnetic (EM) side-channel measurements. This is followed by the work in [36] where the authors utilized EM and a margin-based adversarial attack to recover the structure and the weights for a BNN accelerator. In another attempt Maji et. al. [21] utilized timing and SPA techniques to recover inputs and model for different precisions such as fixed point, floating point and binary NNs. Further, they demonstrated their attacks on multiple micro-controller based devices. The work by Yoshida et. al. [35] demonstrated weight recovery around custom Systolic arrays based NN FPGA implementation. For a timely and excellent survey on the physical side-channel attacks on embedded neural networks, readers may refer to [3], [25], [34].

All of the previous works targeted either micro-controller 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 [15], Intel NCS [13] and NVIDIA's NVDLA [29]. With regards to Intel NCS, three works have been published in literature so far. The first work [33] developed execution time templates using kernel density estimator (KDE) corresponding to Resnet and VGG family of models to find the model used. This is later followed by a cold-boot based attack in [31]. In this case, the authors targeted to recover the model by freezing the RAM on Raspberry Pi. In both the mentioned works, the authors utilized Raspberry Pi as the target for attack with Intel NCS being used for inferencing purpose. The first attempt on an NN ASIC was performed on an Intel NCS in [32]. The authors targeted recovery of model weights using 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. For easier comparison, Table 1 summarizes the state-of-the-art.

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 NVIDIA's 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 demonstrating 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 utilized 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. Further, demonstrating successful recovery of neural network parameters even from a deeply pipelined accelerator which allows parallel execution and is running along with an OS. We effectively show that the leakage corresponding to different parameters is distinctively visible in the power traces and hence, can be easily exploited to recover any CNN model. Using LeNet as the target victim model, we demonstrate very high accuracy of more than 95% in recovering different parameters using trained attack AI models. Even though in this work we used NVDLA as the target, one of the main aim of this work is also to bring attention towards:

- How secure are commercial NN accelerators?
- What are the challenges and possible remedies for the overall experimental setup to analyze such accelerators?
- Does parallel and pipelined execution affect the overall recovery rate?
- How can we efficiently reduce the background noise?
- To what extent, is it possible to recover hyperparameters from a highly optimized FPGA-based NN accelerator?

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 in-depth details about 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 is 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 talk about the NVDLA architecture consisting of its software stack as well as the hardware architecture. The interested readers are referred to [29] for more in-depth details about the architecture.

Figure 1 shows the overall flow of the NVDLA platform. NVDLA natively supports 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

TABLE 1: Overview of state-of-the-art

Attack	Attacked Network	Physical Target	Type of attack	Limitations
Hua et. al. [12]	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]	MLP, CNN	Atmel ATmega328P, ARM Cortex-M3 Microcontrollers	Timing & EM	Hyper-parameters such as kernel size, stride, padding etc. not targeted.
Yu et. al. [36]	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. [21]	Low level operations like RELU, MAC, Multiply etc.	Micro-controllers 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 micro-controllers.
Yoshida et. al. [35]	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. [33]	CNN	Raspberry Pi	Timing Templates	Distinguishing attack applicable only on known models.
Won et. al. [31]	CNN	Raspberry Pi	Cold-boot attack	Not easily ported to other platforms.
Won et. al. [32]	BNN	Intel NCS	CPA	Partial weight recovery with strong assumptions.
Dubey et. al. [9]	BNN	Self designed BNN inference hardware accelerator	CPA	Target platform not publicly used. Accelerator is very specific to their trained network model.

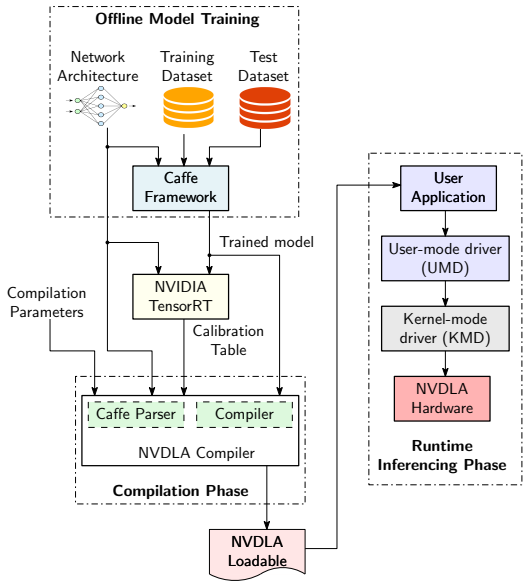


Figure 1: The overall platform flow

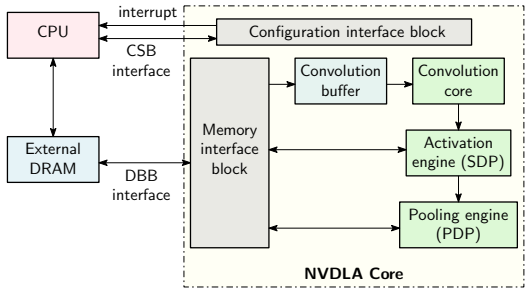


Figure 2: The hardware architecture of NVDLA core

hardware understandable sequence and 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 acts 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 NVDLA core is shown in Figure 2. As can be seen from the figure, NVDLA has a dedicated unit which is used to perform different operations required in a neural network. Convolution core is used to perform the MAC operations required for a convolution and a fully connected layer. The SDP unit is used for bias addition as well as for activation layer. Similarly, 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 which are available in *nv_small* configuration. The *nv_large* or *nv_full* configuration further has support for more features/units such as batch normalization, a dedicated SRAM interface, etc.

2.1.1. NVDLA Convolution Operation. The convolution operation of NVDLA is performed using multiple operations - atomic, stripe, block, channel and group operations. From the attack perspective, atomic and stripe operations are important. Thus, we present details about these operations only.

For the sake of simplicity and explanation, here we assume the kernel size to be 3×3. The weight matrix is represented as below:

$$W = \begin{bmatrix} w_{k0} & w_{k1} & w_{k2} \\ w_{k3} & w_{k4} & w_{k5} \\ w_{k6} & w_{k7} & w_{k8} \end{bmatrix} \quad (1)$$

where w_{k0} denotes first weight byte corresponding to k^{th} kernel. The input image is assumed to be 7×7 and is

represented as:

$$Img = \begin{bmatrix} in_0 & in_1 & in_2 & \dots & in_6 \\ in_7 & in_8 & in_9 & \dots & in_{13} \\ in_{14} & in_{15} & in_{16} & \dots & in_{20} \\ \vdots & & & \ddots & \\ in_{42} & & & & in_{48} \end{bmatrix} \quad (2)$$

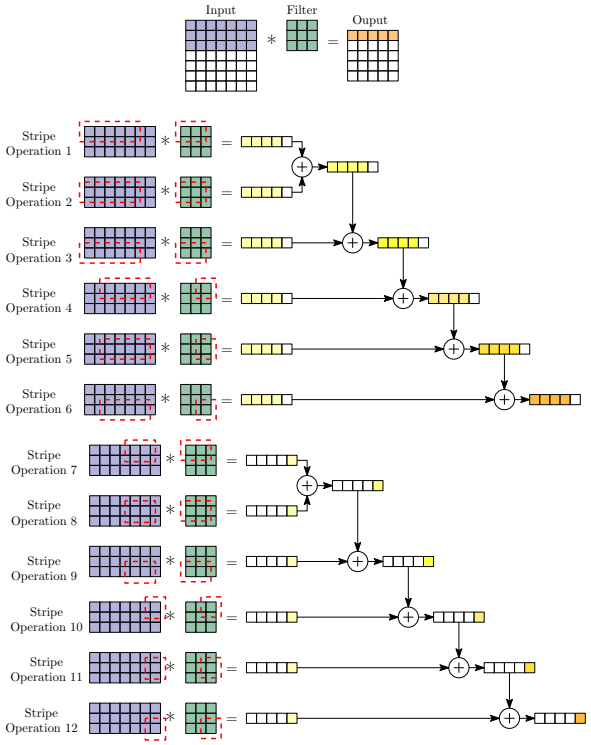


Figure 3: Convolution operation: considering input image of size 7×7 and only one filter of 3×3 to generate one row of output

Figure 3 illustrates how the filter slides on input image across different stripe operations to create the first row of output feature map. A stripe operation comprises of multiple atomic operations. In a stripe operation, the weights are kept stationary throughout whereas input is updated in each clock cycle. Within an atomic operation, the same input is broadcasted to all the MAC units. The weights are updated across different stripe operations. The stripe operations are performed in the MAC Unit, and the generated partial results are accumulated together in the Accumulation Unit of NVDLA. The individual atomic operations (multiply-and-accumulate) being performed at each clock cycle inside a stripe operation are shown in Figure 4. Considering stripe operation 1, the weight and input data selected for this operation will be [w_{00}, w_{01}] and [in_0, in_1, in_2, in_3, in_4] using equations 1 and 2 where k = 0. The computation done at clock cycle 1 can be summarised as ($in_0 * w_{00}$ + $in_1 * w_{01}$). Similar operation is performed for the rest of the clock cycles 2, 3 and 4 with different inputs ([in_1, in_2] at clock cycle 2, [in_2, in_3] at clock cycle 3 and so on). As mentioned before, weight remains stationary during the stripe operation. Note that a stripe of size four results in four partial results corresponding to 4 different output bytes. Further, for stripe operations 1, 2, 3, 7, 8, and 9, two weights are being multiplied with two input bytes whereas for stripe operations 4, 5, 6, 10, 11, and 12 only one weight is multiplied with one input byte. This is because the kernel size is odd (3×3). Hence, in the latter case, the output of stripe operation will be ($in_0 * w_{00}$).

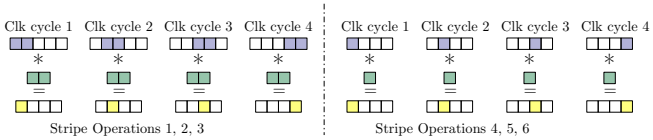


Figure 4: Stripe operation: considering only one filter and a stripe of size 4

Figure 3 and Figure 4 shows the data-flow for both inputs and weights and also the mathematical operation being performed for a single kernel (one MAC instance). But, NVDLA architecture allows multiple MAC instances in parallel. For instance, for NVDLA small architecture, one can use 8 such instances. Each MAC is being utilized to perform convolution with one, resulting in convolution of 8 kernels with the input image in parallel with each other as shown in Figure 5. The partial results corresponding to each kernel are stored in a buffer and accumulated at a later stage.

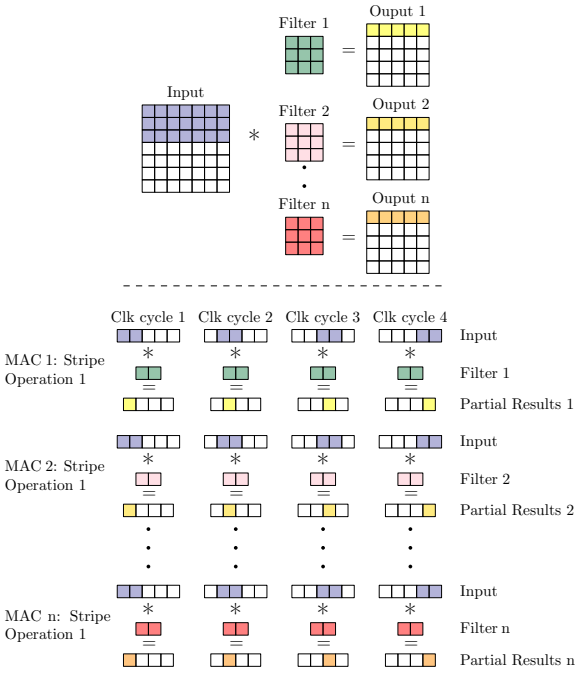


Figure 5: Stripe operation: considering n filters and a stripe of size 4

2.2. Side-Channel Attacks

The seminal work by Paul Kocher in 1996 [17] 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 the secret information (for instance, key). Over the years, it has been shown that any device while performing cryptographic operations leaks data in the form of power consumption [6], [16], [26], electromagnetic emanations [2], [4], [10], [23], timing information [5], [18], 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 it is performing cryptographic operations. A power trace is the current signature produced when a device is performing an operation. There are numerous ways in which the 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) [22], [24]. 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 applied extensively to recover secret information depending on the 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. Further, the attacker is able to capture power traces and observe timing patterns for the different operations. One should note that the operations are executed in a black-box setting. The main target of the attacker is to recover the network structure by identifying different layer types, 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 both in terms of computational power and time for training a neural network customized for a specific application by fine-tuning the parameters. A motivated attacker can attempt to perform network recovery in order to significantly reduce development time and effort, thereby gaining monetary benefits or market advantage.

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 platform. It utilizes Xilinx XC7K160T FPGA as the cryptographic FPGA and has 162,240 logic cells. Out of the three configurations - *nv_small*, *nv_large* and *nv_full* provided by NVIDIA, *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. We faced many challenges as discussed below in order to successfully develop the setup. 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. One needs to manually enable the correct drivers 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. Also, 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 32-bit Microblaze system. However, the drivers provided by NVIDIA access memory using 64-bit address which conflicts with a 32-bit based Microblaze. To address this issue, we modified the provided drivers 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*. But, 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) Apart from porting the source code for Microblaze system, the setup for NVDLA requires loading the loadable files for inferencing using a SD-card or using SSH/SCP through ethernet. The SASEBO-GIII does not have the 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 PC to the board.

These changes allowed for 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 commonly used neural networks. The network is trained offline using Caffe framework [14] and then compiled into a loadable file using NVDLA compiler. As shown in Figure 1, 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 [19]. 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 28 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 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. 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. But, as NVDLA is a highly optimized accelerator, multiple operations execute in parallel and in a pipelined manner. For instance, 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, the data transfer from DRAM also sometimes happen 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 so many different components interacting with each other and thus, 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. Further, 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 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.

In general, we observed that using higher order filters provide 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 filter 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 NVDLA accelerator is significantly larger than the Microblaze.

3.4. 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 large memory in the oscilloscope. This is necessary as lower sampling rate results in layers diminishing around the noise level due to low accuracy, which makes it quite challenging to distinguish between NN layers 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 as well as maintain the high sampling rate of around 2GS/s.

3.5. 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 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 overall pattern of the captured traces.

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
 1   slen // denotes the length of downsampled
     trace
 2 i := 0
 3 final_samples := slen/2
   /* denotes the length of samples considered
      in a window                              */
 4 samples_per_block := tlen/final_samples
 5 current_block_pos := 0
 6 while i < (final_samples * 2 - 2) do
 7     j := 0
 8     W_max := INT_MIN
 9     W_min := INT_MAX
       /* find maximum and minimum in the
          current window                       */
10     while j < samples_per_block do
11         current_value := T[current_block_pos +
            j]
12         if current_value > W_max then
13             W_max := current_value
14         if current_value > W_min then
15             W_min := current_value
16         j := j + 1
       /* save the datapoints                  */
17     if i % 2 == 0 then
18         S[i] := W_max
19     else
20         S[i] := W_min
       /* slide window with overlapping half
          points                               */
21     current_block_pos = current_block_pos +
        (samples_per_block/2)
```

We first evaluated different known approaches such as averaging, decimation, interpolation, etc. But the obtained results 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.

3.6. Final measurement setup

An Agilent DSO6034A oscilloscope was used to capture the traces from the SASEBO board. The measurement setup is shown in Figure 6. 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. 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 Figure 7. Hence, we used this combination to capture all the traces in our experiments. Also, a DC block was used to remove the 1V bias voltage from the FPGA VCCINT signal.

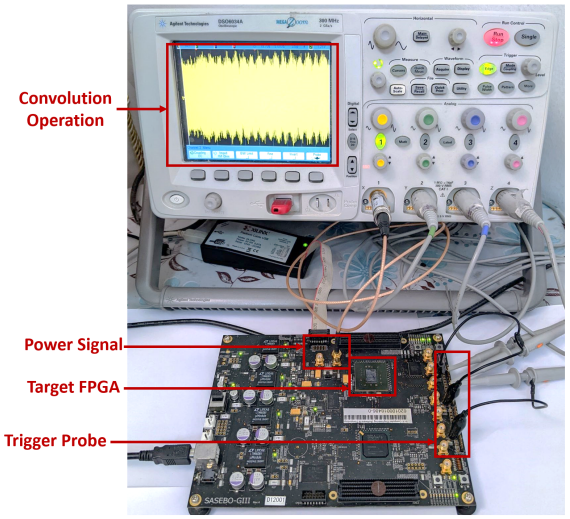


Figure 6: Overall measurement setup

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

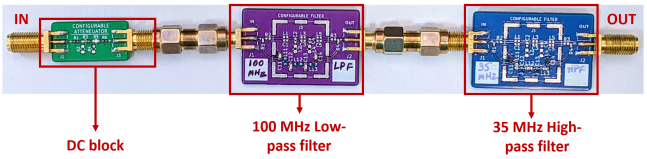


Figure 7: Filter setup

traces except for the zoomed versions. The downsampling was performed using Algorithm 1. As per the NVDLA documentation, only ReLU and PReLU are supported for *nv_small* configuration. But, out of these two, only ReLU activation is supported in *nvdla_compiler* [28] for parsing. Hence, we only focus on identifying and differentiating between convolution, pooling and fully connected layer and their respective parameters.

4.1. Identifying different number of layers

The first and foremost target in structure recovery is to identify the number of layers in the neural network model. The complete model takes about 0.5 seconds to finish execution using NVDLA. Due to the memory limit of the oscilloscope, we captured the trace for this part in multiple chunks as described in Section 3.5. In Figure 8, 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.

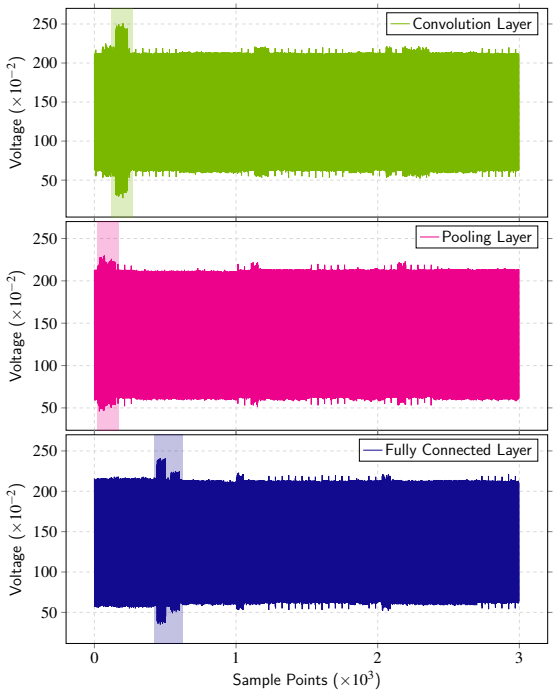


Figure 8: Truncated Linux execution window = 5ms

The higher peaks is the time when the corresponding layer is being executed in the hardware. Whereas, we believe 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 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 small (a few microseconds).

4.2. Distinguishing different layer types

Figure 9 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 are 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 Figure 10. 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. Whereas, in case of a fully connected layer, the peaks are always continuous and fine-grained.

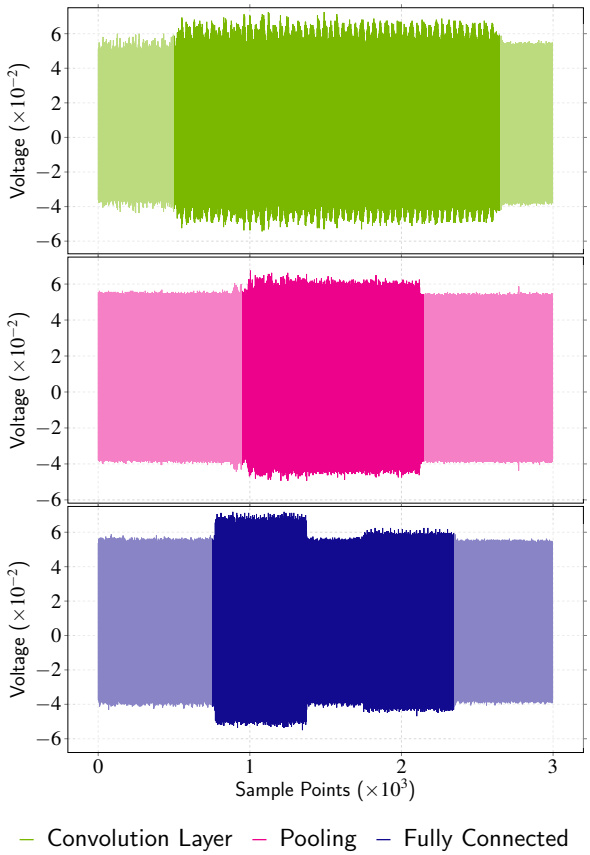


Figure 9: Different layers

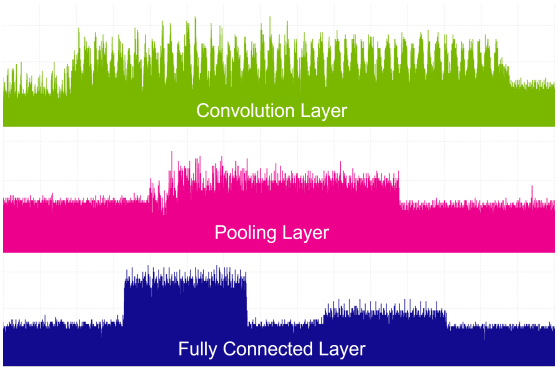


Figure 10: Zoomed in view of different layers

4.3. Reverse engineering of the convolution hyper-parameters

4.3.1. Kernel size. The next target is to identify the kernel size. As the most commonly used kernel sizes are 3×3, 5×5 and 7×7. So, in our attack, we focused on these three kernel sizes only. The obtained power trace is shown in Figure 11. The points a, b and c mark the end of convolution operation for different kernel sizes.

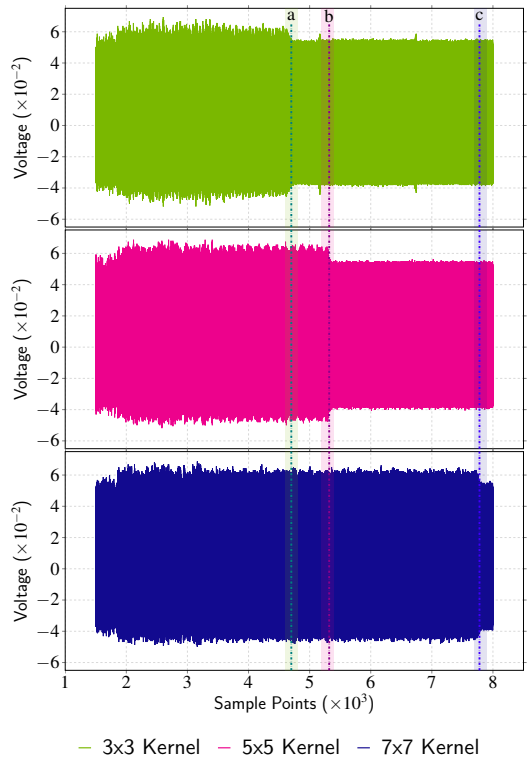


Figure 11: Identifying kernel size

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 convolution operation with a 5×5 kernel and point c denotes the end of convolution with a 7×7 kernel. This demonstrates that there is a significant difference in execution time for different kernel sizes.

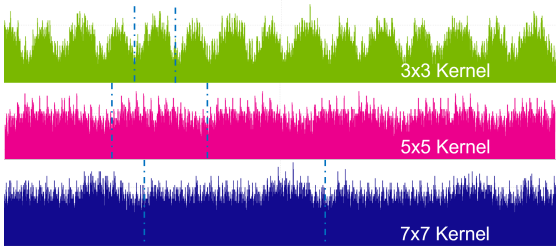
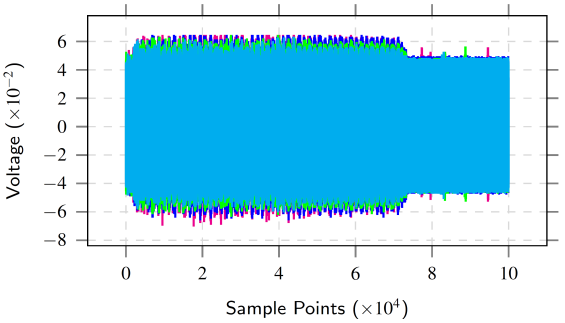


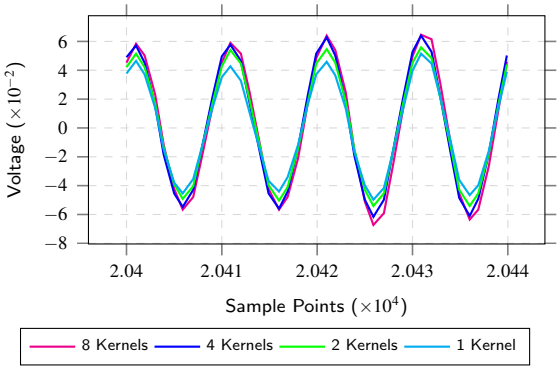
Figure 12: Zoomed in width difference for different kernel sizes

The larger kernel size requires more number of stripe operations. As a result, the width of individual peaks visible in Figure 11 is different for different kernel sizes. This is demonstrated in Figure 12. 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.



(a) Number of Kernels (3×3)



(b) Number of Kernels Zoomed (3×3)

Figure 13: Number of kernels

In our experiments, we observed that the power leakage is additive as shown in Figure 13. The figure shows power consumption traces when the number of kernels executed in parallel are 1, 2, 4 and 8 respectively. One can clearly see that there is a difference in the power leakage depending on the number of kernels. For better clarity, we also show a zoomed in version of the trace (Figure 13b). Using this observation, one can build a power template and use it to easily guess the number of kernels being executed in parallel.

NVDLA small allows execution of only 8 kernels in parallel. Hence, if the number of kernels in the model are 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. The individual batches are also visible in the power trace when they are executed one after another. We demonstrate this for different kernels in Figure 14 where we show the execution of 1 kernel versus 20 kernels. One can see that the time required to finish execution of 1 kernel is repeated three times for 20 kernels.

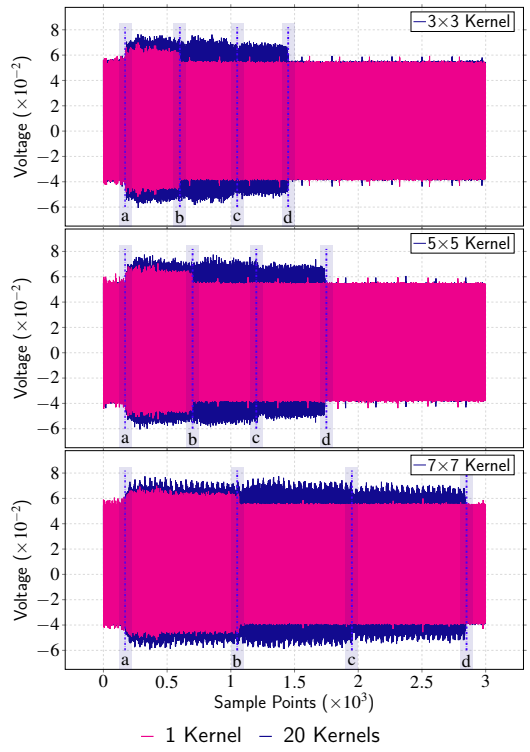


Figure 14: Effect of batch size and kernel size

The points a-b, b-c and c-d in the figure mark three batch execution for the current model. As there are three batches, it can be concluded that a maximum of 24 kernels are there in the architecture. In order to get the precise number of kernels, the strategy discussed for Figure 13 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 which we exploited earlier in identifying the different kernel sizes. Another interesting thing to note is that for the 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 instance, if the kernel size is 3×3, then the padding size can only be less than 3. Hence, we show the obtained power trace for only valid padding cases. Figure 15 shows the difference in padding possible for a 3×3 kernel size. One can note that as the padding size increase, the number of individual peaks also increases. Further, the width of the peak does not change as the kernel size is not changing.

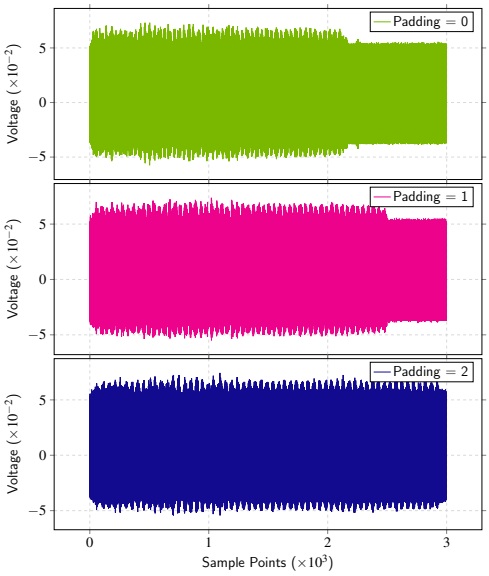


Figure 15: Padding for 3x3 kernel

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 Appendix A (Figure 21). One interesting thing to note is that the increase in number of peaks or the 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 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. Figure 16 shows the difference between different strides for a 3×3 kernel. Further, the stride is applied when sliding from both 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 clearly visible from the power trace as well. When stride=3, the convolution execution period is significanty lower when compared to that of stride=1.

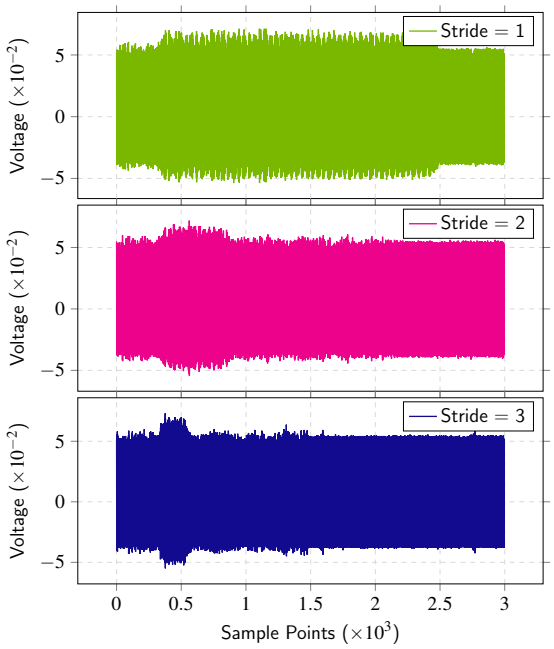


Figure 16: Stride for 3x3 kernel

4.4. Reverse engineering pooling layer

Apart from recovering different parameters for the convolution layer, we also did some experiments to observe leakage patterns for a pooling layer to identify its kernel size and stride. As 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 is using 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 same as in a convolution layer.

4.4.1. Kernel size. The traces obtained for two different kernel sizes 2×2 and 3×3 are shown in Figure 17. It is quite evident that power consumption required when kernel size is 3×3 is consistently more compared to when it is 2×2. This might be because more number of pixels are being processed in a single clock cycle in case of a 3×3 kernel as compared to a 2×2 kernel.

4.4.2. Stride. Figure 18 shows the traces for different strides. It is interesting to note that the power consumption

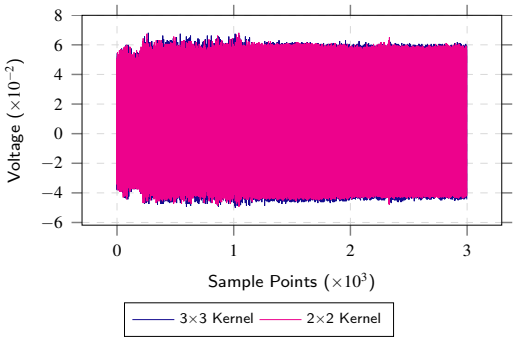


Figure 17: Pooling kernel size

required when stride=1 is more compared to when stride=2. This is true for both a 2×2 and a 3×3 kernel. We believe this is because when stride=2, the amount of processing is less as compared to stride=1.

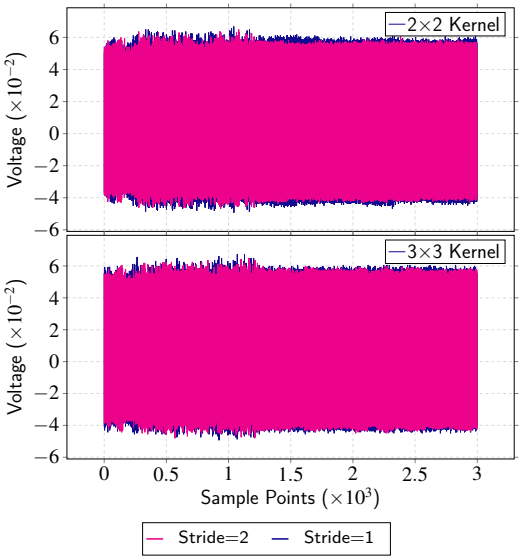


Figure 18: Distinguishing stride for pooling layer

4.5. AI-assisted automated attack flow

Figure 19 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 the trained models for inferencing on 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 for training the attack CNN models in step ❹. Steps ❶-❹ constitute the initial profiling phase of the attack.

In step ❺ and ❻, the target victim model is used for actual application inference and the final outcome is provided

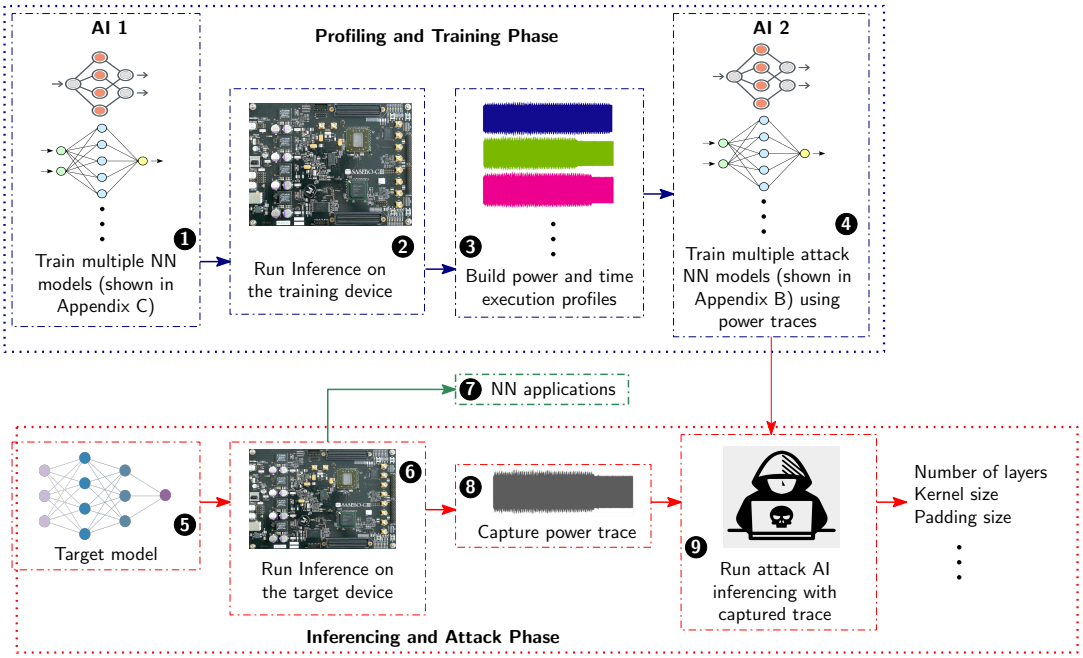


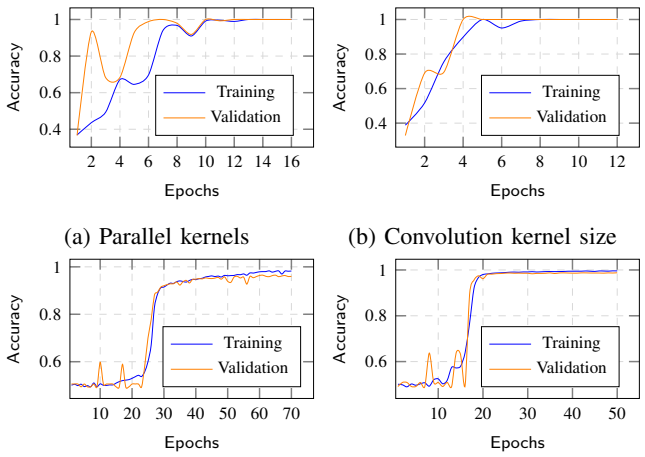
Figure 19: Overview of the AI-assisted attack flow.

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 four attack models for reverse engineering kernel size, number of kernels executing simultaneously in a convolution layer, distinguishing pooling layer type (Maximum or Average) and pooling kernel size. The attack models were trained using Keras and Tensorflow backend. The details of respective model architecture are provided in Appendix B. As mentioned in section 4.5, 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 for LeNet network.

Figure 20 shows the accuracies for different trained models. For all the experiments, we used 20% of the training set 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 are clearly visible, as a result the model is trained faster with very few traces and epochs. Whereas, 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.



(a) Parallel kernels (b) Convolution kernel size

(c) Pooling layer type (d) Pooling kernel size

Figure 20: Training and validation accuracy for different attack models

Table 2 presents 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 the LeNet is quite high. This validates the effectiveness of our trained attack models for reverse engineering the victim model structure and its parameters.

TABLE 2: Summary of the test accuracies.

Ref. as	Layer	Architectural Parameters	Our model		LeNet	
			Test Set	Acc.	Test Set	Acc.
AM1	Conv	Parallel kernels	1500	100%	100	100%
AM2	Conv	Kernel size	1500	100%	100	100%
AM3	Pool	Type (AVG or MAX)	2000	96.7%	100	95%
AM4	Pool	Kernel size	1000	99.0%	100	99%

Even though we demonstrate the automated attack using four network parameters, the same methodology can be applied to perform recovery of rest of the parameters. This is because most of the differences in the parameters can be clearly observed in the traces as discussed in section 4 and can be trained using AI. 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 make any attempt to recover 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.

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	–

Further, recovery of inputs and weights require a different form of attack like Differential Power Analysis. In this work, we have not targeted such attacks and left the possibility of recovering these parameters as possible future extension.

6. Further Discussions

- **Porting our setup and attack to other NN accelerators.** As discussed in Section 3, apart from integrating the accelerator in the architecture, there are many other difficulties which can result in unsuccessful recovery and thus, biased evaluation. The currently available NN accelerators such as Intel NCS, Google's TPU are available in the form of a USB stick, so we believe integration might not be an issue. The rest of the approaches discussed in this paper for signal filtering, full trace capture and downsampling can be used for evaluating these accelerators as well. Further, one can experiment with a similar attack strategy using EM for measuring the power traces and observing the difference.
- **Applicability on other real-world models.** Since the side channel leakage is clearly visible in the power traces, and most of the network parameters are recoverable, thus, the attack techniques demonstrated using LeNet in this work can be directly applied to recover other real world models such as AlexNet, SqueezeNet etc. as well, especially as they do not add any more attack complexity compared to LeNet.
- **Possible side-channel countermeasures.** Typical side-channel countermeasures like masking are used to protect secrets such as inputs, weights etc. [7]–[9], but they do not protect the overall operation and information such as number of layers, type of layers, etc. There has been some progress in protecting the structure using obfuscation techniques [20] such as layer widening, layer branching etc. But, 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 commercial widely deployed 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 for the developed tools at https://github.com/ after publication.

References

[1] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 515–532, USA, 2019. USENIX Association.

[2] Vincent Carlier, Hervé Chabanne, Emmanuelle Dottax, and Hervé Pelletier. Electromagnetic side channels of an fpga implementation of aes. In *CRYPTOLOGY EPRINT ARCHIVE, REPORT 2004/145*. Citeseer, 2004.

[3] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. Side channel attacks for architecture extraction of neural networks. *CAAI Transactions on Intelligence Technology*, 6(1):3–16, 2021.

[4] Elke De Mulder, Pieter Buysschaert, SB Ors, Peter Delmotte, Bart Preneel, Guy Vandenbosch, and Ingrid Verbauwhede. Electromagnetic analysis attack on an fpga implementation of an elliptic curve cryptosystem. In *EUROCON 2005-The International Conference on" Computer as a Tool"*, volume 2, pages 1879–1882. IEEE, 2005.

[5] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *International Conference on Smart Card Research and Advanced Applications*, pages 167–182. Springer, 1998.

[6] Xiaoyi Duan, Qi Cui, Sixiang Wang, Huawei Fang, and Gaojian She. Differential power analysis attack and efficient countermeasures on present. In *2016 8th IEEE International Conference on Communication Software and Networks (ICCSN)*, pages 8–12. IEEE, 2016.

[7] Anuj Dubey, Afzal Ahmad, Muhammad Adeel Pasha, Rosario Cammarota, and Aydin Aysu. Modulonet: Neural networks meet modular arithmetic for efficient hardware masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 506–556, 2022.

[8] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Bomanet: Boolean masking of an entire neural network. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[9] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 197–208. IEEE, 2020.

[10] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 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*, pages 1857–1874. ACM, 2017.

[11] Xiaolu Hou, Jakub Breier, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Physical security of deep learning on edge devices: Comprehensive evaluation of fault injection attack vectors. *Microelectronics Reliability*, 120:114116, 2021.

[12] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Intel. Intel Movidius Neural Compute Stick. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-movidius-neural-compute-stick.html.

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[16] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[17] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[18] Francois Koeune, Jean-Jacques Quisquater, and Jean-Jacques Quisquater. A timing attack against rijndael. 1999.

[19] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[20] Jingtao Li, Zhezhi He, Adnan Siraj Rakin, Deliang Fan, and Chaitali Chakrabarti. Neurobfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 248–258. IEEE, 2021.

[21] Saurav Maji, Utsav Banerjee, and Anantha P. Chandrakasan. Leaky nets: Recovering embedded neural network models and inputs through simple power and timing side-channels—attacks and defenses. *IEEE Internet of Things Journal*, 8(15):12079–12092, 2021.

[22] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.

[23] Adam Matthews. Low cost attacks on smart cards: the electromagnetic sidechannel. *Next Generation Security Software, Sept*, 2006.

[24] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92. Springer, 2000.

[25] Maria Mendez Real and Ruben Salvador. Physical side-channel attacks on embedded neural networks: A survey. *Applied Sciences*, 11(15), 2021.

[26] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Investigations of power analysis attacks on smartcards. *Smartcard*, 99:151–161, 1999.

[27] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. Power side-channel attacks on bnn accelerators in remote fpgas. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):357–370, 2021.

[28] NVDLA. NVDLA PReLU Issue. https://github.com/nvdla/sw/issues/16#issuecomment-384517936,https://github.com/nvdla/sw/issues/32, 2018.

[29] NVIDIA. NVIDIA Deep Learning Accelerator. http://nvdla.org/primer.html.

[30] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 393–406, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Arindam Basu, and Shivam Bhasin. Deepfreeze: Cold boot attacks and high fidelity model recovery on commercial edgeml device. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[32] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Arindam Basu, and Shivam Bhasin. 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*, pages 111–114, 2021.

[33] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Shivam Bhasin, and Arindam Basu. Time to leak: Cross-device timing attack on edge deep learning accelerator. In *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4. IEEE, 2021.

[34] Qian Xu, Md Tanvir Arafin, and Gang Qu. Security of neural networks from hardware perspective: A survey and beyond. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 449–454. IEEE, 2021.

[35] 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 *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[36] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 209–218. IEEE, 2020.

Appendix A. Padding

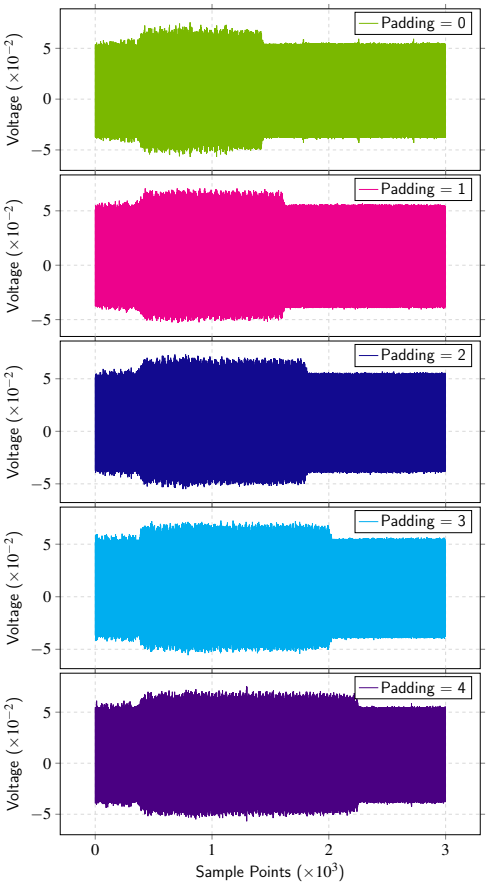


Figure 21: Padding for 5x5 kernel

Appendix B. Attack Models Architecture and Configuration

TABLE 4: Summary of the training configuration.

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

TABLE 5: Model architecture for Attack Model 1 (AM1)

Layer Type	Channels	Filter Size	Stride	Activation
Conv	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	32	2x2	2x2	-
Conv	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	64	2x2	2x2	-
Conv	128	3x3	1x1	LReLU (slope = 0.1)
MaxPool	128	2x2	2x2	-
FC	128	-	-	LReLU (slope = 0.1)
FC	3	-	-	Softmax

TABLE 6: Model architecture for Attack Model 2 (AM2)

Layer Type	Channels	Filter Size	Stride	Activation
Conv	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	32	2x2	2x2	-
Conv	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	64	2x2	2x2	-
FC	128	-	-	LReLU (slope = 0.1)
FC	3	-	-	Softmax

TABLE 7: Model architecture for Attack Model 3 and 4 (AM3 and AM4)

Layer Type	Channels	Filter Size	Stride	Activation
Conv	32	3x3	1x1	LReLU (slope = 0.1)
MaxPool	32	2x2	2x2	-
Conv	64	3x3	1x1	LReLU (slope = 0.1)
MaxPool	64	2x2	2x2	-
Conv	128	3x3	1x1	LReLU (slope = 0.1)
MaxPool	128	2x2	2x2	-
FC	128	-	-	LReLU (slope = 0.1)
FC	2	-	-	Softmax

Appendix C. Models List

Each model in Table 8 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 8: Trained models architecture for profiling and generating datasets

Model #	Operation	Layer Name	Number 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	1×1	0×0
2	Convolution	Convolution + ReLU	2	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
3	Convolution	Convolution + ReLU	4	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
4	Convolution	Convolution + ReLU	8	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
5	Convolution	Convolution + ReLU	20	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
6	Convolution	Convolution + ReLU	1	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
7	Convolution	Convolution + ReLU	20	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
8	Convolution	Convolution + ReLU	1	7×7	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
9	Convolution	Convolution + ReLU	20	7×7	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
10	Convolution	Convolution + ReLU	4	3×3	2×2	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
11	Convolution	Convolution + ReLU	4	3×3	3×3	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
12	Convolution	Convolution + ReLU	4	5×5	1×1	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
13	Convolution	Convolution + ReLU	4	5×5	2×2	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
14	Convolution	Convolution + ReLU	4	5×5	3×3	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
15	Convolution	Convolution + ReLU	4	5×5	4×4	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
16	Convolution	Convolution + ReLU	4	5×5	5×5	0×0
	Pooling	Max pooling	1	2×2	1×1	0×0
17	Convolution	Convolution + ReLU	4	3×3	1×1	1×1
	Pooling	Max pooling	1	2×2	1×1	0×0
18	Convolution	Convolution + ReLU	4	3×3	1×1	2×2
	Pooling	Max pooling	1	2×2	1×1	0×0
19	Convolution	Convolution + ReLU	4	3×3	1×1	3×3
	Pooling	Max pooling	1	2×2	1×1	0×0
20	Convolution	Convolution + ReLU	4	5×5	1×1	1×1
	Pooling	Max pooling	1	2×2	1×1	0×0
21	Convolution	Convolution + ReLU	4	5×5	1×1	2×2
	Pooling	Max pooling	1	2×2	1×1	0×0
22	Convolution	Convolution + ReLU	4	5×5	1×1	3×3
	Pooling	Max pooling	1	2×2	1×1	0×0
23	Convolution	Convolution + ReLU	4	5×5	1×1	4×4
	Pooling	Max pooling	1	2×2	1×1	0×0
24	Convolution	Convolution + ReLU	4	5×5	1×1	5×5
	Pooling	Max pooling	1	2×2	1×1	0×0
25	Convolution	Convolution + ReLU	4	3×3	1×1	0×0
	Pooling	Max pooling	1	3×3	1×1	0×0
26	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	2×2	2×2	0×0
27	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	3×3	1×1	0×0
28	Convolution	Convolution + ReLU	1	3×3	1×1	0×0
	Pooling	Max pooling	1	3×3	2×2	0×0