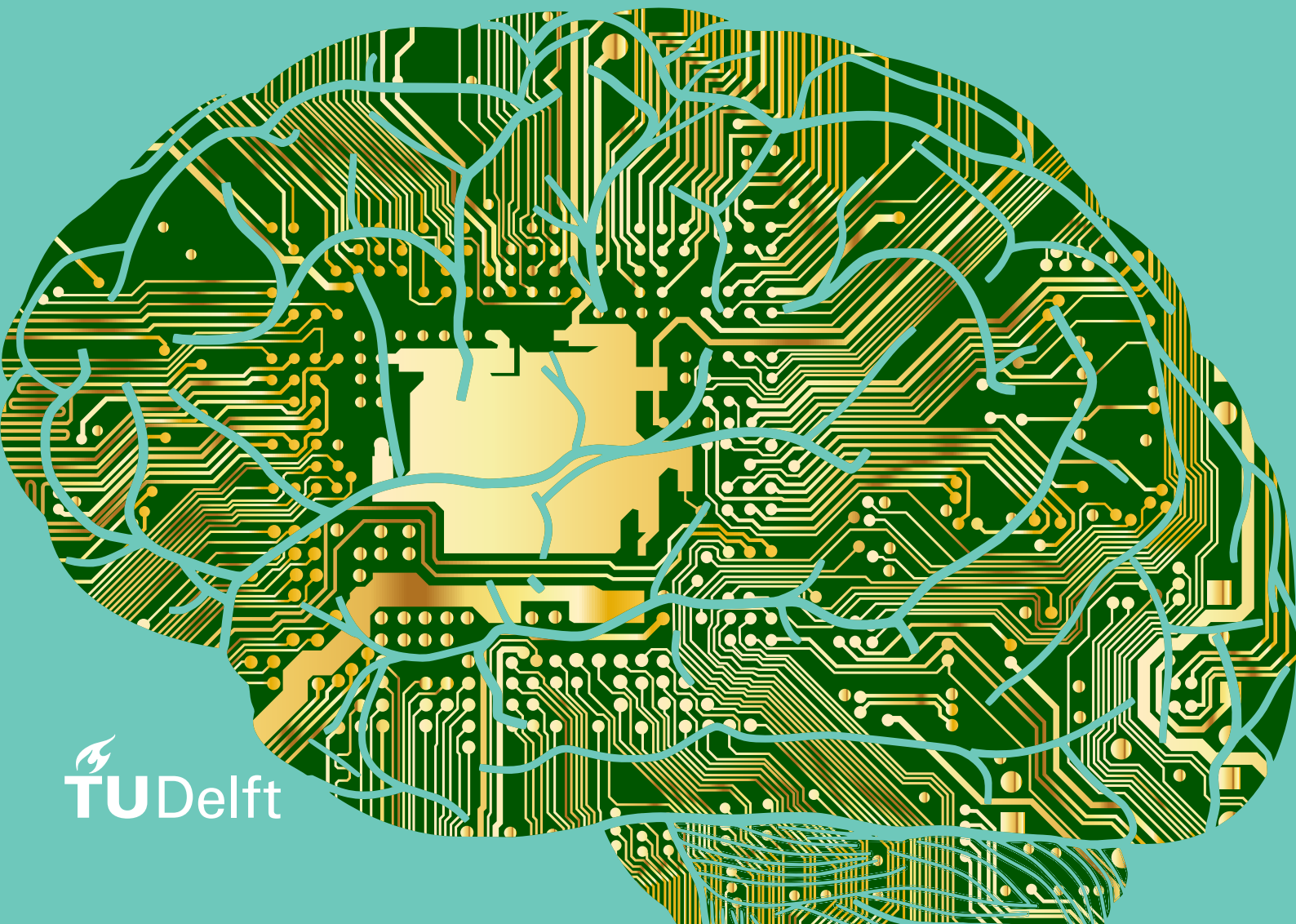
An Analysis of Deep Learning Based Profiled Side-channel Attacks

Custom Deep Learning Layer, CNN Hyperparameters for Countermeasures, and Portability Settings

Rico Tubbing



TU Delft

An Analysis of Deep Learning Based Profiled Side-channel Attacks

Custom Deep Learning Layer, CNN Hyperparameters for Countermeasures, and Portability Settings

by

Rico Tubbing

to obtain the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Thursday December 19, 2019 at 10:00 AM.

Thesis committee:	Dr. S. Picek,	TU Delft, supervisor
	Dr. C. Doerr,	TU Delft
	Dr. P. K. Murukannaiah,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Abstract

A side-channel attack (SCA) recovers secret data from a device by exploiting unintended physical leakages such as power consumption. In a profiled SCA, we assume an adversary has control over a target and copy device. Using the copy device the adversary learns a profile of the device. With the profile, the adversary exploits the measurements from a target device and recovers the secret key. As SCAs have shown to be a realistic attack vector, countermeasures have been invented to harden these kinds of attacks.

In the last few years, deep learning has been applied in a wide variety of domains. For example, convolutional neural networks have shown to be effective for object recognition in images and recurrent neural networks for text generation. In the side-channel analysis domain, deep learning has shown to be successful. Up until recently, no deep learning layer existed that was specifically designed for SCAs. In this work, we analyze this layer, called the spread layer, and demonstrate the flaws of this layer. We improve the flaws and show the spread layer does not enhance the performance of SCAs. Additionally, we show there is no need to develop a deep learning layer specifically for SCAs on unprotected implementations.

For implementations where countermeasures are present, literature demonstrated that convolutional neural networks are the most successful. However, for both the masking and random delay countermeasure, little is known about the influence of the kernel size and depth of the network. In this work, we illustrate that increasing the kernel size and depth of the network both increase the attack efficiency for the random delay countermeasure. For the masking countermeasure, we demonstrate that higher kernel sizes and shallow networks perform the best.

Additionally, in this work, we consider a portability setting where the probe position has been changed in between the measurements of the profiling and attack measurements. Here, we show that the probe position causes a typical deep learning SCA to be ineffective. We introduce a normalization method such that the attack becomes effective, and show this method enables the attack to perform as expected.

For countermeasures this however different. We know that CNNs work really well for SCAs where a random delay countermeasure is present. However little is known about the influence of the hyperparameters. We experiment with the kernel size and depth of a CNN , and show that increasing both results in more efficient attacks.

We do the same for the masking countermeasure, but see that the CNN does not provide good results. We believe the CNN is too complex, which we believe because MLP networks are significantly more efficient.

Portability experiment with a setting where the probe position has been changed in between measurements of the profiling and attack traces. Difference in the traces We see that normalizing the profiling and attack traces separately resolves the issue, and show that we can perform a standard side-channel attack

Acknowledgements

This thesis in front of you is the result from months of hard work. It has been an enjoyable but also a rough road in which I have learned a lot about both research and myself. As this thesis is a result from me and the people around me, I would like to thank the people who have been important for me during this research project.

First of all, a special thanks goes out to my daily supervisor, Stjepan Picek. You have guided, inspired, and pushed me to work hard and efficient such that we, together, can reach the highest. Although, sometimes for you it might not have seen like hard work as your timing to check on us was I would say special, since you somehow managed to check on us whenever we took a coffee break. And of course, I would like to thank you for given me with the opportunity for my research visit in Singapore, I had never imagined this would even be possible.

This brings me to my time in Singapore. I would like to thank my supervisor in Singapore, Shivam, for providing me with this opportunity as well. Thanks for your excellent guidance during these months, but also showing us the cool places in Singapore, and of course the pool party. Thanks to Daan, Johannes, and Romain for exploring Singapore and surroundings with me.

Back home, the lunches and the apparently a bit too loud coffee breaks with the threat intelligence group have been a real pleasure to me. For these I would like to thank Christian, Daan, Harm, Hugo, Sandra, Tim, Vincent, and Wilko. I really enjoyed the discussions about everything and nothing at the same time. Next to the threat intelligence group, I like to thank the crypto group. I want to thank Zeki for allowing me to take the PETs course when I was stressed out. Additionally, I am thankful for the lunches with your students, Chi, Gamze, and Ozzy.

Furthermore, I would like to specifically thank Daan and Wilko. Both helped me to complete my thesis with meaningful discussions about our theses, but also the less serious conversations, jokes, and laughter.

I would also like to express my gratitude for my friends at home, Donny, Joris, Lars, Lex, Rutger, Stefan, Timon, and Wouter. In the evenings, and weekends you brought pleasure and laughter which allowed me to put my mind on somethings else. Because of this I could come back with more focus and energy the next day.

And finally, I would like to express my special thanks of gratitude to my brothers, Frank and Tom, and my parents, Gerard and Yvonne. My parents have always supported me even through the times I thought I messed up once too many. You cared for me, give me time to figure out what I like, and allowed me to pursuit my interests in computer science. Because of you I was able to finish my Bachelors degree, start my masters, and we are now so close to finishing it. Without you this would have not been possible.

Rico Tubbing
Delft, December 2019

Contents

1 Introduction

In 1965, a spy from the British Military Intelligence Agency, the MI5, bugged the Egyptian embassy located in London. Dressed up as a telephone technician, a British spy infiltrated the embassy and secretively planted a microphone near a rotor-cipher machine. Back then, such machines were used for the encryption and decryption of messages, and the 'key' for encryption or decryption would be the initial settings of the machine's wheels. Each morning the embassy's cipher clerk would reset the wheels' settings. The sound produced by these adjustments was loud enough to be picked up by the planted microphone, which allowed the MI5 to deduct the initial settings of two or three of the in total seven wheels. With this additional knowledge, the keyspace was reduced significantly such that the British had sufficient computational power to break the encryption, and thus were able to spy on the embassy's communication [51, 56].

The attack performed by the British is now known as a *side-channel attack* (SCA). An SCA exploits some unintended physical leakage while the system performs operations on sensitive values. By analyzing the obtained physical leakages an adversary can recover the sensitive values. In the attack performed by the British, the physical leakage was the sound produced by adjusting the wheels, and the sensitive values were the initial settings of the wheels. SCAs have shown to be effective using various physical channels, such as sound [2], electromagnetic radiation [1], power consumption [26], and accelerometer data [9].

Nowadays, rotor-cipher machines are no longer used for secure communication, instead, we rely on cryptographic mechanisms implemented in software and hardware to securely communicate. One of these mechanisms that is used extensively, is called the *Advanced Encryption Standard* (AES) and since its introduction, no feasible key recovery attack is known. In 2015, the best known mathematical attack requires $2^{126.01}$ key attempts, which takes several years with nowadays computational power [50]. However, this attack only considers the cipher from a mathematical perspective and does not consider that AES is implemented or runs on a physical chip (hardware or software). Similar to the rotor-cipher machines, electrical chips have some physical leakage such as power and electromagnetic radiation. Depending on the implementation of AES, the chip's physical leakages makes it vulnerable to SCAs. Since the discovery of side-channel attacks, the community has invented methods to protect from these attacks. In side-channel analysis, these methods are called *countermeasures* and can be implemented in either software or hardware. Generally, countermeasures are split into two categories: masking and random delay.

With the rise of IoT devices and smart-cards, the attack surface of side-channel attacks has grown. These devices are vulnerable to side-channel attacks because typically the security is neglected to reduce costs and keep the chips small. Since the security is neglected and AES is commonly used in these devices, it is an interesting target for side-channel attacks.

The history of side-channel attacks started with the discovery of *Simple Power Analysis* (SPA), in which an adversary visually analyzes an observed leakage to deduce sensitive information. Later, the side-channel analysis community found *Differential Power Analysis* [26] and *Correlation Power Analysis* [6], in which multiple observations are statistically analyzed to recover the sensitive data. These attacks require only access to a target device and are known as *non-profiled* attacks. In a *profiled* attack, an adversary has access to the target device and a copy of it. To perform the attack, the adversary creates a profile of the copy device's observations and uses the profile to predict the sensitive values given observations from the target device. To create the profile, an adversary learns a statistical model.

To generate a profile of a device, *Template Attack* (TA) has historically been used. It has been shown that TA is the most powerful attack from a theoretical perspective [8]. However, as *Machine Learning* obtained quite some attention in various domains, the side-channel analysis community has successfully shown that ML is a viable approach for profiled side-channel attacks. Moreover, it has been shown that an ML approach can outperform TA in some settings [30].

As ML techniques seem to be promising, *Deep Learning* (DL) techniques have become popular for side-channel attacks. Recent works have shown that DL is an effective tool for side-channel attacks, even in the presence of countermeasures [33]. Next to this, DL usually requires little to no pre-processing, such as feature selection, to perform successful attacks. Because of the recent success with deep learning side-channel attacks, we investigate deep learning for profiled side-channel attacks in this thesis.

1.1. Problem Statement

In other domains than side-channel analysis, DL has been successfully applied and shown to increase performance. For example, a significant performance boost was found by the image recognition domain when using DL techniques. Moreover, the largest performance boost was found by using convolutional neural networks (CNNs). Such networks exploit domain-specific characteristics that allow them to improve performance. In the side-channel analysis domain, we know that CNNs perform well for both unprotected and protected implementations. Notably, CNNs excel when attacking an implementation with a random delay countermeasure. It thus seems that some deep learning layers can eliminate the effect of a countermeasure. Recently a new deep learning layer has been proposed for side-channel analysis which was shown to improve the attack efficiency on unprotected implementations. As there has been no work done that analyzes this layer, we will perform an analysis of this layer, and pose the question if it makes sense to create a layer specifically designed for side-channel attacks on unprotected implementations. If so, this might open many possibilities for more successful attacks which in turn increase the quality of security assessments. For example, this could lead to deep learning layers designed to eliminate a countermeasure's effect.

Despite the shown success of deep learning side-channel attacks, it suffers from some disadvantages. To perform successful attacks, a suitable network architecture has to be configured. There are a significant amount of imaginable architectures, and only a limited amount of these architectures provide good performance for side-channel attacks. Even with a suitable architecture, deep learning still requires careful choice of parameters since the wrong configuration results in a non-successful attack. Additionally, deep learning networks are considered as black-box, meaning we do not understand the internal workings of the network. This toughens the choice of architecture and parameters, which explains why related works attempt various configurations to obtain the best results. Current research has found some general directions for specific SCA datasets, however, knowledge on how to configure architectures is still lacking. For countermeasures, we know that CNNs can achieve state-of-the-art performance. However, a common baseline of parameters is unknown. In this work, we investigate the design of CNN architectures and their respective parameters.

Another problem in this domain is that many works do not consider a realistic setting, in which profiling and attack traces are obtained from the same device. Recently, literature has shown that in these settings the attack efficiency is overestimated. These works showed that when using different devices for profiling and attacking, the devices have different characteristics, making the attack harder. The literature has proposed some methods to deal with this, however, still work is required to gain more understanding.

In general, we aim to improve the performance of side-channel attacks on AES, such that we help to improve the quality of security assessments. This is important because by doing so we improve the security of AES implementations. To do so we provide guidelines that have shown to work in a variety of settings. Furthermore, the guidelines improve the understanding of SCA with deep neural networks. Although we do not understand the decision making in a neural network, the guidelines show what configurations work well in specific settings.

In summary, we identify the following problems:

- No extensive research has been conducted on specially designed layers for side-channel analysis.
- The design of network architectures has a significant influence on the attack success, but there is limited knowledge about the architecture's configuration for side-channel analysis.
- Literature has shown that in realistic settings, the profiling measurements are different from the attack measurements.

In this thesis, we pose the question if there is a need for a deep learning layer designed for side-channel attacks on unprotected implementations. Furthermore, we wonder what the influence of a subset of CNNs' hyperparameters is for specific countermeasures. Lastly, we ask if the repositioning of the probe in-between measurements influences the attack efficiency and if so what can be done to improve the attack efficiency.

1.2. Contributions

So far, the problems we have identified are far from trivial and require knowledge about side-channel analysis, AES, and deep learning techniques. Furthermore, the training of deep neural networks is computationally heavy and time-consuming, limiting the possibilities to experiment with a wide variety of settings. Despite these difficulties, our contributions are as follows:

- Show that special designed deep learning layers for non-protected implementations are not necessary, and the already known layers are sufficient.
- Provide an evaluation of the design for CNN architectures, and provide a baseline for datasets in which the random delay countermeasure is present.
- Provide an evaluation of the design for CNN architectures, and provide a baseline for datasets in which the masking countermeasure is present
- A novel approach to normalize training and test traces such that SCAs are effective in a portability setting.

1.3. Outline

The rest of this thesis is outlined as follows. First, the background knowledge required for a basics understanding of the performed work is introduced. Subsequently, we will discuss the related literature in the field of side-channel attacks using deep learning, identify where current research is lacking, and formulate our research questions. Then we will continue with the search for a deep learning layer specifically designed for side-channel attacks. In the following chapter, we discuss the influence of various hyperparameters of CNN architectures for specific countermeasures. Furthermore, we investigate the effect of normalization in a portability setting, show that a typical approach results in ineffective attacks. Therefore, we introduce a novel method to normalize the attack traces, which improves the attack efficiency significantly. Finally, we answer the research questions using the knowledge from the conducted experiments, discuss the limitations of the performed work, and finally discuss possible future work.

2
Background

In this chapter, we discuss the background knowledge required for the research presented in this thesis. First we discuss the encryption standard AES, the target of our attacks. Then we will provide an overview of machine learning and deep learning techniques. In the next section, we introduce side-channel attacks, in which we discuss non-profiled and profiled side-channel attacks. Additionally, we give an overview of the datasets used for the experiments. Finally, we describe the technical details of the conducted experiments.

2.1. AES

AES is a family name used to describe three ciphers, which each have different key sizes, that is AES-128, AES-192, and AES-256. AES is a symmetric key algorithm which means that the same key is used for both encryption and decryption. In AES, encryption and decryption are applied on a block of bytes, which makes AES a block-cipher. We describe the encryption function of AES as $E_k(m)$, where m is the message and k the key. Similarly, we describe the decryption function $D_k(c)$, with the cipher-text c and key k. Since AES is a symmetric key algorithm, the following equation is correct:

$$D_k(E_k(m)) = m.$$

In this work, we only discuss AES-128, other key lengths are disregarded. For simplicity, we will only discuss the encryption process in this section, but for completeness, we mention that the decryption of a ciphertext is obtained by applying the inverse of the encryption process.

AES exists of ten rounds, and in each round, the same sequence of operations is performed. In each round, a round-specific key, called the *round key*, is used for encryption. The round keys are derived from the master key and are the same size as the master key, the derivation of the round keys is called KeyExpansion. Rijndael's key schedule is used to calculate round keys [37], but we will not go in further detail how this key derivation scheme works.

The operations performed during the encryption process are applied on a four by four state matrix, each state in the matrix represents a single byte. Initially, the encryption function uses each byte of the plaintext to fill each entry of the state matrix. Thus, each byte of the plain text represents one entry in the state matrix. Similar to the state matrix, the round key is used to generate a matrix called the round key matrix. It is similar to the state matrix, except that it contains the current round key. The state matrix is represented as a, and $a_{i,j}$ is an entry in the state matrix where i and j are the row and columns in the state matrix respectively. Similarly, k represents the round key matrix, and $k_{i,j}$ represents a single entry in the round key matrix.

After the initialization of the cipher, the encryption process starts. AES consists of four operations that are repeatably used, called AddRoundKey, SubBytes, ShiftRows, and MixColumns. In the following sections, we will discuss these operations and how they form the AES encryption algorithm.

2.1.1. Operations

The AddRoundKey operation performs a bitwise xor between each entry of the state matrix and each corresponding entry of the round key matrix. Mathematically, this is described as:

$$\text{AddRoundKey}(a,k) = \forall i \forall j : a'_{i,j} \leftarrow a_{i,j} \oplus k_{i,j}. \tag{2.1}$$

In Figure 2.1 a visualization of the AddRoundKey operation on the state matrix is shown. In this figure, the highlighted parts on the left part of the image represent the selected entry of the state matrix and the corresponding entry of the round key matrix. A bitwise xor is performed on these two entries, and the result is stored in the state matrix.

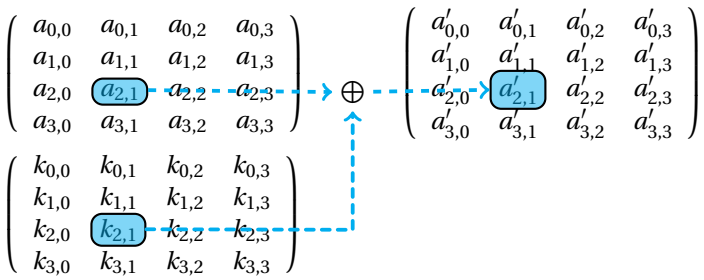


Figure 2.1: The AddRoundKey performs the xor operation for each entry of the state matrix with the corresponding entry of the round key matrix. The result is the updated state matrix, shown as the matrix on the right side in the figure.

The SubBytes function performs substitution of each entry in the state matrix $a_{i,j}$ using a look-up table. The look-up table is called the S-box in AES. The SubBytes operation provides AES with its non-linearity. Mathematically this operation is described as:

$$\text{SubBytes}(a) = \forall i \forall j : a'_{i,j} \leftarrow \text{S-box}[a_{i,j}].$$

In Figure 2.2, a visualization of the SubBytes operation is shown. In this figure, the highlighted part of the state matrix on the left is selected and used to perform a look-up in the S-box. The result is the updated state matrix, shown as the right matrix in this figure.

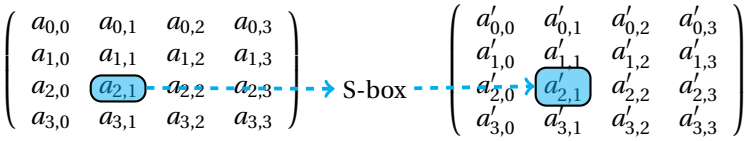


Figure 2.2: The SubBytes operation performs for each entry of the state matrix a look up, the result of this lookup is stored in the updated state matrix shown here as the matrix on the right.

The ShiftRows function performs a cyclic shift to the left on each of the rows of the state matrix, except for the first row. The second row shifts one position to the left, the third shifts two, and the fourth shifts three. In Figure 2.3, this operation is visualized. In the first row, there is one highlighted bar, in the second row, the highlighted bar is divided into two, such that the bar with the rounded corner on the left is similar to the original state matrix. This is similar for the other rows.

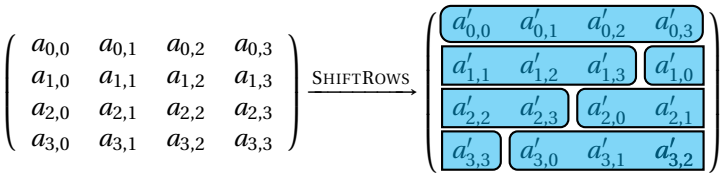


Figure 2.3: The ShiftRows operation shift each row, except for the first row, in a cyclic manner by a fixed offset. The second row is shifted one position to the left, the third is shifted two, and the third row is shifted three positions to the left.

The MixColumns function diffuses the columns. It performs matrix multiplication in the finite field GF(s^8) on the columns as shown in Figure 2.4. This figure shows that the four entries of a column are mingled which results in a new column a'.

$$\begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} \xrightarrow{\text{MIXCOLUMNS}} \begin{bmatrix} a'_{0,i} \\ a'_{1,i} \\ a'_{2,i} \\ a'_{3,i} \end{bmatrix} \leftarrow \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix}, \text{where } 0 \leq i \leq 3$$

Figure 2.4: The MIXCOLUMNS operation combines each column of the state matrix into a new column using matrix multiplication over the finite field GF(2^8).

We have discussed all the operations used to perform AES encryption. Here we will explain in which order AES applies these operations to encrypt a plaintext. In Algorithm 1 pseudo-code of the encryption algorithm of AES is shown. Initially, AES deduces the round keys from the master key using the KEYEXPANSION algorithm, which are then stored for later use. After this, the ADDROUNDKEY operation is applied and the first round starts. In the first round the operations SUBBYTES, SHIFTROWS, MIXCOLUMNS, and ADDROUNDKEY are applied sequentially. This sequence of operations is repeated nine times. The last round differs from the previous rounds as it does not apply the MIXCOLUMNS operation. This operation is skipped because it does not add any security if it would have been added.

Algorithm 1 An overview of the AES encryption algorithm.

```
roundKeys ← KEYEXPANSION(key)
state ← plaintext
state ← ADDROUNDKEY(state, roundKeys[0])
for i = 1, i < 10, i++ do
    state ← SUBBYTES(state)
    state ← SHIFTROWS(state)
    state ← MIXCOLUMNS(state)
    state ← ADDROUNDKEY(state, roundKeys[i])
end for
state ← SUBBYTES(state)
state ← SHIFTROWS(state)
state ← ADDROUNDKEY(state, roundKeys[9])
ciphertext ← state
```

2.2. Machine Learning

Machine learning (ML) is the science in which a computer is learned to solve a given task without the computer being explicitly programmed to solve the given task [5]. Typically, ML algorithms are divided into two categories: *supervised* and *unsupervised* learning. Supervised ML requires a dataset in which input and output pairs are known. In an unsupervised ML setting only the input is known to the algorithm. In this work, we will only discuss ML in a supervised setting.

Generally, supervised ML consists of two phases: the training and test phase. In the training phase, an ML algorithm aims to learn patterns from training examples. Usually, the training examples are referred to as the *training data*, which consists of a set of input-output pairs. Here we will refer to a single input as a *feature vector* $\vec{x} = (x_1, x_2, \ldots, x_n)$, where n is the number of features. Typically, ML learns by iteratively minimizing a cost function, which indicates the costs of the predictions and the ground truth. More formally, the training phase can be defined by learning a function $f : \mathbb{X} \rightarrow \mathbb{Y}$, which maps the input domain $\mathbb{X}$ to the domain $\mathbb{Y}$. The configurations used to train a model are referred to as *hyperparameters*, which are distinct from the *parameters* of an ML model. The hyperparameters are configured before the start of the learning algorithm, while the parameters are adjusted during the learning phase.

After the training phase, the performance of the model is evaluated, this phase is referred to as the testing phase. In the testing phase, the model is used to perform the given task on unseen data. The unseen data is referred to as the test data. The key difference between the training and testing data is that the testing data has not been used during the training phase.

The separation of training and test data is necessary to create an accurate evaluation of the performance

of the learned model. If there is no separation, and the same data is used for the training and testing phase, an inaccurate evaluation of the performance of the model is generated, because the model is learned using the training data. As a result, a situation could occur where the model can correctly predict the output for the training data, but not for the testing data. The learned model is thus not able to generalize and fails to predict the ground truth of the test data. This phenomenon is well-known in the ML domain and is usually referred to as *overfitting*. Overfitting generally occurs when the model is too powerful for the given task, i.e. the model has too many parameters, and as consequence, the model is capable of identifying the feature vectors from the training data individually, such that it can learn the ground truth. An example of when overfitting occurs is when a non-linear model is learned on linear data. In this setting, there is a high probability that the model learns feature vectors of the training data individually and thus fails at producing correct results for unseen data.

Another well-known problem of ML techniques is called *underfitting*, which is the opposite of overfitting. The learned model is not able to correctly predict the ground truth given the training and test data. Here we say that the model is not able to learn anything from the training data. Generally, underfitting occurs when a model is not powerful enough to capture the patterns in the training data. An example of such a situation is when a linear model is learned on non-linear data. In this situation, the model will not be able to learn the training data correctly to predict the ground truth for both the train and test data.

If we would configure the hyperparameters of a model to perform well on the test data, we introduce a bias to the model. Therefore, we require an additional data set to be used to configure the hyperparameters, this data set is called the *validation set*. Additionally, the validation set can be used to detect underfitting or overfitting. This is particularly useful when the training phase takes a long time. The problems can be recognized by analyzing the value of the cost function on the validation set. If a problem occurs, the training phase can be stopped and the hyperparameters can be adjusted, and valuable training is prevented from going to waste. In subsection 2.4.4 we will briefly discuss a technique that can automatically detect and will quit training if a model is overfitting during the training phase.

So far we have discussed the training and testing phase, and the problems during training. Here, we will discuss the problems ML solutions are typically used for. Typical ML tasks are regression and *classification*. In a regression problem, the output of a model is a continuous value. A typical example of a regression problem is estimating the salary of a person given features such as age, sex, job, etc. In a classification problem, the output of a model is not a single value, but a set of values, each representing a single class. A well-known classification problem is object recognition in images, in which a model is trained to recognize objects in images. In this work, we will only discuss classification problems. In a classification problem, the model is usually referred to as a classifier, we will use the notation and classifier interchangeably.

To evaluate the performance of a classifier, several metrics have been suggested by the machine learning community. The most well-known metrics are *accuracy*, recall, and precision. In this work, we only use accuracy as a metric (see subsection 2.5.4). Trivially, the accuracy indicates the percentage of correctly predicted outputs of the classifier. Equation 2.2 shows how accuracy is calculated.

$$Accuracy = \frac{\#correct\ predictions}{\#predictions} \tag{2.2}$$

2.3. Deep Learning

Deep learning is a branch of the ML domain which is based on artificial neural networks (ANNs). An ANN is said to be an engineered system inspired the brain Since the brain is able to perform and solve complex tasks, such as image recognition and language processing, it is widely thought that gaining an understanding of how the brain learns, could provide the scientific community insights into how to develop complex neural networks that are capable of performing on a similar level as humans [15].

In this section, we will first discuss a classical neural network called multilayer perceptron and how these types of networks calculate the predictions given an input. To do so first the building blocks of these types of networks are discussed as well. Subsequently, we will discuss convolutional neural networks and the properties of these networks.

2.3.1. Multilayer Perceptron

A multilayer perceptron (MLP) is one of the simplest types of neural networks. As all deep learning neural networks, MLP consists of an input layer, at least one hidden layer, and an output layer. The input layer is the input of the network, and similarly, the output layer is the network's output. The hidden layers are defined

as the layers between the input and output layers. Each layer consists of a configured amount of neurons, which each applies mathematical functions on its input to calculate the output. The input layer has the same number of neurons as the length of the feature vector, while the output layer has the same number of neurons as the total number of classes. In each layer, mathematical operations are applied to the input of the layer, and the output is passed on to the successive layer in the network, which is either a hidden layer or the output layer.

In an MLP network, each neuron's output is used by each neuron of the successive layer. In Figure 2.5, we depict an example MLP network, where a node represents a neuron and the edges the information flow. In this figure the input feature vector $\vec{x}$ exists of six features, x_1 till x_6, two hidden layers with four neurons in each layer, and two output neurons, y_1 and y_2.

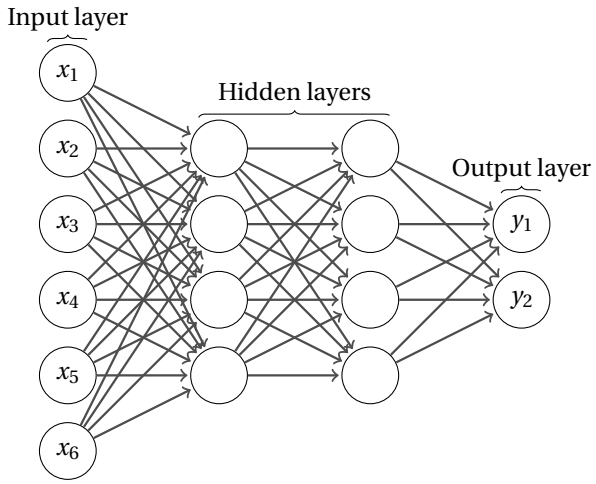


Figure 2.5: A multilayer perceptron neural network with six input features, two hidden layers with each four neurons, and two output neurons. The edges represent the weighted output of a neuron.

A neuron's output is calculated by the activation of the sum of the weighted outputs of the previous layer and the bias term. The output of a neuron is also called the activation of a neuron. Mathematically we describe the activation of the kth neuron in the lth layer as shown in Equation 2.3.

$$x_k^l = \sigma\left(\sum_{i=0}^{N^{l-1}} w_{k,i}^l \cdot x_i^{l-1} + b_k^l\right) \tag{2.3}$$

In this equation, the superscript describe a layer's index, and the subscript the neuron's index. Furthermore, σ is the activation function, w a neuron's weight, b a neuron's bias, x a neuron's output, and N the number of neurons in a layer. To make this more clear, Equation 2.3 is visualized in Figure 2.6 and shows how the output of a neuron is calculated. In this figure, the output is calculated by taking the activation of the sum of the weighted inputs from the neurons x_1 to x_n, and the bias.

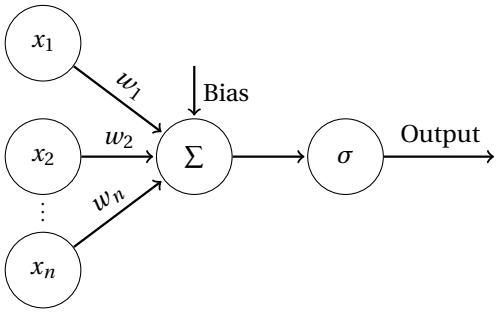


Figure 2.6: The calculation of the output of a perceptron visualized.

Neural networks are commonly trained on GPUs since the calculations can be performed as matrix multiplications and additions. This makes the training phase considerably faster than on a CPU. Equation 2.3 can

be described as matrix multiplication and addition with the following equation:

$$x_k^l = \sigma\big((w_k^l)^T \cdot x^{l-1} + b^l\big). \tag{2.4}$$

In Figure 2.5, each edge represents the weighted input of the neuron it is connected with. From this figure, we can determine the total amount of parameters used in the network. The amount of parameters is commonly used to compare network architectures. In the first hidden layer there are four neurons, witch each a bias and six connections with the previous layer, thus there are $4 * 6 + 4 = 28$ parameters in this layer. Similarly, the number of parameters can be determined for the following layers. In total there are $28+20+10 = 58$ parameters in this network. If we imagine that we have a large MLP network, with more neurons in a layer, from the calculation it is clear that larger networks consist of many parameters, which is one of the reasons why neural networks are so powerful.

2.3.2. Activation Functions

In Equation 2.3 we have described how a neural network calculates the output. However, we have not discussed the activation function σ. The activation function provides neural networks with non-linearity, which allows them to learn not only linear problems but also non-linear functions. Since the output of the activation function is a neuron's output, this output is also referred to as the activation. In this section, we will discuss the most used activations.

Historically, sigmoid is the activation function that was used in all neural networks. The *sigmoid* activation function is defined as

$$\text{SIGMOID}(x) = \frac{1}{1+e^{-x}}. \tag{2.5}$$

Sigmoid suffers from the vanishing gradients, which is not a desirable property for a neural network. Vanishing gradients prevent the network from learning new patterns. This problem occurs since it has a horizontal asymptote and as a result: for a large x sigmoid outputs a value close to the zero, while for a significant larger x' sigmoid will output a similar value [28]. Another problem mentioned in [28], the output of sigmoid is not on average close to zero. To fix this problem it has been suggested to use hyperbolic tangent (also known as *tanh*) as an activation function. It is defined as:

$$\text{TANH}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{2.6}$$

but it can also be expressed using sigmoid:

$$\text{TANH}(x) = 2\text{SIGMOID}(2x) - 1. \tag{2.7}$$

Thus, tanh also suffers from vanishing gradients since it is a scaled and shifted version of sigmoid.

Nowadays, the most common activation function that is used in neural networks, is called rectified linear unit or also called *ReLU*. ReLU is described as:

$$\text{RELU}(x) = \max(0, x). \tag{2.8}$$

Thus, ReLU takes the maximum of zero and the activation of a neuron.

The main advantage of ReLU is that it suffers significantly less of vanishing gradients in comparison to sigmoid and tanh, and therefore networks that use ReLU can learn better. Another advantage of ReLU is the computational costs, it requires only comparisons and no additional calculations. ReLU is simple and performs extremely well in various settings.

There is one more activation function that is frequently used, which is called softmax. This activation function has a different purpose in networks than the other discussed functions. Its main purpose is to assign probabilities to the output classes of a network and is thus used as the last layer of the network. Without this function, the output of a network would be harder to reason about. The softmax function is defined as:

$$\text{SOFTMAX}(\vec{x}^{l-1})_i = x_i^l \leftarrow \frac{e^{x_i^{l-1}}}{\sum_{k=1}^{K} e^{x_k^{l-1}}}. \tag{2.9}$$

Here, l refers to the layer for which the output is computed, and $l-1$ the previous layer, i is in the index of a neuron. The result of this function is a vector of probabilities of each class, thus, the sum of the output vector sums to one. Next to this, Equation 2.9 shows that a larger input will yield a larger probability.

2.3.3. The Learning Process

So far we have discussed how a neural network computes a prediction given an input, and how activation functions make neural networks non-linear such that they can compute complex functions. Here, we will discuss how a neural network gradually learns. As previously discussed, each neuron has an associated bias and associated weights for each neuron of the previous layer. Neural networks learn by iteratively adjusting the weights and biases each epoch. For now, we assume that in an epoch a single forward pass and a backward pass are performed. We define the forward pass as the calculation of the predictions on training or test data-set. In the backward pass, the weights and biases of the model are adjusted to improve the predictions of the network. Usually, the backward pass makes use of a technique called backward propagation. Backward propagation requires us to define a cost function, which indicates the error between the predicted output and the ground truth. Backward propagation determines in which direction each weight and bias should be adjusted to minimize the cost function, the direction the parameters should be adjusted is also known as the gradient. Backward propagation is possible because the model is differentiable which means that each weight and bias is differentiable as well. This makes it possible to determine in which direction individual weights should be adjusted to reach a local or global minimum [45].

Since the gradient only indicates in which direction a step should be performed to minimize the cost function and not the magnitude of the step, another hyperparameter is introduced to control the magnitude of the step. This hyperparameter is called the learning rate. The adjustment for a weight w is denoted as Δw and is calculated by the formula shown in Equation 2.10.

$$\Delta w = \eta * \frac{\partial C}{\partial w} \tag{2.10}$$

In this equation, η is the learning rate, C the cost function, and w the weight. The new weight is calculated by adding Δw to w. If the learning rate is set too high this causes backward propagation to jump over the local minimum and thus might never reach the local minimum. On the other hand, if the learning rate is set too low this causes the network to learn too slow, which results in long training times. No single learning rate suits every problem, and thus the best learning rate is found by experimental validation.

A technical problem occurs if we attempt to perform backward propagation over the entire data-set. To do so, the entire dataset and the network's parameters are required to be stored in memory. If either of these two is large, storing this might not be possible for the GPU memory. As a result of this physical limit, it is unfeasible to calculate the cost function over the entire data set. A solution to this problem is to iteratively supply a subset of the feature vectors to the network, this subset and its size are referred to as a batch and the batch size respectively. The batch size thus defines the size of the subset of feature vectors for a single forward pass. Since the data set is split into multiple batches, it is necessary to perform multiple forward and backward passes. We now define a single epoch as the forward and backward passes to process the entire dataset. This means that after each forward pass the weights and biases are adjusted (in the backward pass) such that it fits better over the just supplied batch. Similar to the learning rate, there is no magic number for the batch size that is suitable for all problems and thus the best batch size is usually experimentally discovered.

The classifications problems we discuss in this report are called one-of-many classification. In such a problem there are C classes, in which there can be only one correct. *One-hot encoding* is used to encode the ground truth values. One-hot encoding turns a class value into a vector $\overrightarrow{y} = [c_0, c_1, \ldots, c_{C-1}]$, where each entry represents a class and c_i is equal to one if it is the ground truth, otherwise zero. The cost function, or also called lost function, is the error between the predicted classes and the ground truth. Typically, for one-of-many classification problems, *categorical cross-entropy* is the loss function. It is calculated by the equation shown in Equation 2.11, where y_i and $\hat{y}_i$ of class i are the ground truth and predicted value respectively.

$$L(y, \hat{y}) = -\sum_{i=0}^{C} y_i * log(\hat{y}_i) \tag{2.11}$$

In summary, a network is trained by iteratively supplying the training data and corresponding ground truth values. Using backward propagation, we can determine the direction the parameters have to be adjusted to minimize the cost function.

2.4. Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of deep neural networks and are one of the most popular types of networks that are used in the deep learning community. Their performance is one of the reasons

why CNN architectures are commonly used. Since 2012, CNNs have emerged in many applications that are capable of outperforming previously state-of-the-art solutions. Well-known applications in which CNNs can reach new state-of-the-art performance are image classification, image recognition, and natural language processing.

The first application in which a CNN outperforms any other existing solutions is presented in [27]. In this work, the authors trained a CNN on the ImageNet dataset and competed in the ILSVRC-2012 competition. They achieved a top-5 error rate of 15.3%, while the second-best, a non-CNN solution, was only able to reach an top-5 error rate of 26.2%. Ever since this contribution, CNNs have gained in popularity and extensive research have been done on these types of networks. CNNs have shown to be capable of improving state-of-the-art performance in various domains [23, 43].

In this section, we will discuss how CNNs work, and what makes them so powerful in comparison to other types of neural networks. To do so, we will first introduce the convolutional layer, the heart of each CNN. Afterward, we will introduce additional layers that are commonly used in CNN architectures. Finally, we will describe CNN architectures that are often found in the literature using the introduced layers.

2.4.1. The Convolutional Layer

The convolutional layer is the most essential layer of CNNs. In a convolutional layer, *kernels* slide over the input, the output is calculated by performing element-wise multiplication between a window and a kernel. A kernel is a set of learnable parameters. In Figure 2.7, an example visualization of applying a convolution layer on a 4x4 matrix is shown. In this figure, a 4x4 matrix is convolved using a 2x2 kernel, the colored squares in this figure highlight the windows that are used to calculate the corresponding colored output. Since it is not possible to highlight all input-output pairs, only three input-output pairs are highlighted.

In a convolutional layer, the size of the kernels can be configured and is thus a hyperparameter, we refer to this hyperparameter as the kernel size. The image classification community use kernels that are between 3x3 and 7x7 [27, 46]. In chapter 3, we will discuss the influence of the kernel size in side-channel analysis.

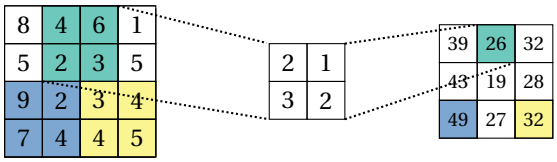


Figure 2.7: The result of applying a 2x2 convolutional filter on a 4x4 matrix, resulting in a 3x3 matrix. The highlighted input-output pairs are the result of performing element-wise multiplication on the highlighted part of the input and the filter. The other entries of the output matrix are calculated similarly.

CNNs have a property that allows them to outclass other NNs, they are shift-invariant: the exact position of a meaningful feature in the feature vector is not important, as long as it is roughly the same a CNN is capable of recognizing this feature. Because the kernels slide over the input an exact position of a meaningful feature is not important. Thus, if a kernel is able to detect a pattern in the feature vector, this kernel will also detect this pattern at a different position in the feature vector.

The intuition behind a convolutional layer is that the kernels are able to learn patterns in the data. Since the kernels slide over the input, a convolutional layer is capable of extracting similar patterns of the activations locations with the same set of kernels. Usually, convolutional layers are stacked on each other. The key idea behind stacking these layers is that the first layers are able to learn small patterns in the data, while the latter layers use these patterns to detect more complex patterns. For example, in the image recognition domain, the first few layers are able to recognize lines and corners, and the latter layers are able to detect objects such as tables and chairs [34].

The example in Figure 2.7 shows that the input matrix, a 4x4 by matrix, is convolved to a smaller 3x3 matrix. The decrease in dimensionality is an immediate result of applying a convolutional layer. To prevent a decrease in dimensionality, zero-padding is typically applied to the input. There are several ways in which this can be done. For example, for the matrix shown on the left in Figure 2.7, adding zeros to the left and top of the matrix will create in a 5x5 matrix, which will yield a 4x4 output matrix by applying a 2x2 kernel. In Figure 2.8 we depict an example where we apply padding to a 4x4 matrix which finally produces the same dimension matrix when applying a 2x2 kernel. Typically, same-padding is applied to the input data, which tries to insert an equal amount of zeros to all possible locations, the top, bottom, left, and right of the input matrix.

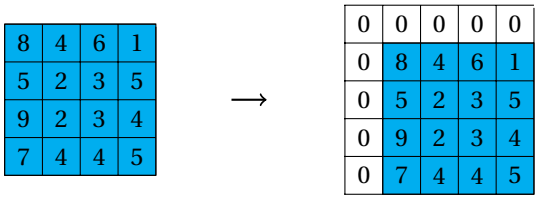


Figure 2.8: Applying padding to a 4x4 matrix shown the left, with the output matrix shown on the right.

Another hyperparameter that can be adjusted is called the stride, which defines the magnitude of the step a kernel slides over the input. Increasing the stride is usually done to increase the receptive field. The receptive field is a term from the biological domain, where it denotes the 'portion of sensory space that can elicit neuronal responses when stimulated' [3]. Similar in the deep learning domain, the receptive field is the region of space that is affected by a neuron [32].

Another advantage of CNNs is that the number of parameters of a convolutional layer does not depend on the length of the input vector, but it depends on the kernel size and the number of kernels. If we compare this property to an MLP network, where the number of parameters depends on the number of neurons of a layer and the successive layer in the network then an MLP network has significantly more parameters. For example, assume we have a feature vector that has 1 000 features, in an MLP network this would mean that each neuron of the following layer requires 1 000 parameters. Comparing this with a convolutional layer with 128 5x5 kernels, there are only 3 200 parameters.

In summary, the convolutional layer applies multiple kernels to its input. By sliding the kernels over the input a CNN is able to learn patterns in the data.

2.4.2. Pooling

As we have briefly discussed, a convolutional layer calculates the output by applying numerous kernels. As a consequence, the spatial dimension of the data increases significantly. To counter the explosion of the spatial dimensionality of the data, a *pooling* layer is introduced. The key purpose of a pooling layer is to compress the spatial dimensionality to increase training times. Similar to a convolutional layer, in a pooling layer, a window slides over the input and produces an output. Here the features in a window will be mapped to a smaller spatial dimension, usually a single feature.

The two most well-known pooling layers are called *max-pooling* and *average-pooling*. In Figure 2.9 and Figure 2.10 the process of applying max pooling and average pooling, respectively, is shown. As the names suggest, max-pooling works by selecting the maximum value in the window, while average-pooling outputs the average in the window.

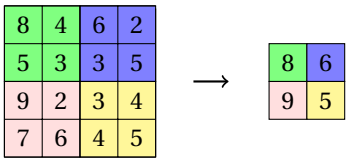


Figure 2.9: A visualization of the max pooling operation on a 4x4 input matrix. The kernel size is 2x2, and the stride is set to two. Each highlighted window of the output matrix is the result of selecting the maximum value of the corresponding highlighted square of the input matrix.

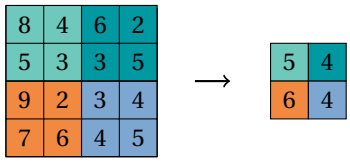


Figure 2.10: A visualization of the average pooling operation on a 4x4 input matrix. The kernel size is 2x2, and the stride is set to two. Each highlighted square of the output matrix is the average of the corresponding highlighted square of the input matrix.

2.4.3. Batch Normalization

In [22], the authors mention that the internal covariate shift deteriorates the learning capabilities of neural networks. The internal covariate shift is defined as the change of distribution of the activations while a model is learning. In simpler words, a successive layer does not account for the shift of distribution, which toughens the learning process. To reduce the effect of this problem the authors introduce a new layer called *batch normalization* (BN). As the names suggest, a BN layer is a layer that normalizes the activations of a batch, and next to this it also shifts the normalized value by some learnable parameters. Aside from the reducing covariance shift, BN also allows for higher learning rates, which in turn speeds up the training phase, and

reduces the problem of exploding and vanishing gradients. Since the introduction of the BN layer in 2015, the deep learning community has adopted this layer in many architectures and the addition has shown to be successful in many architectures [49, 53].

To normalize the activation of a batch, the BN layer first calculates the mean $\bar{x}$ and variance σ^2 of a batch. Then, the mean and variance are used to normalize the activations. After the normalization, a scaling operation is applied, this operation has two learnable parameters, β and γ. Equation 2.12, Equation 2.13, and Equation 2.14 show how the mean, variance, and the normalization plus scaling is applied respectively. In these equations m is equal to the batch size, k is the index of a neuron for which BN is applied, i is the index of a neuron in the batch, $x_{k,i}$ is the activation of a neuron k and the ith element in the batch. The output of a single neuron of the batch normalization is $y_{k,i}$.

$$\bar{x}_k = \frac{1}{m}\sum_{i=1}^{m} x_{k,i} \tag{2.12}$$

$$\sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{k,i} - \bar{x}_k) \tag{2.13}$$

$$y_{k,i} = \beta + \gamma * \frac{x_{k,i} - \bar{x}_k}{\sqrt{\sigma_k^2}} \tag{2.14}$$

Typically, BN layers are placed at either two positions in the network architecture, either before or after the activation function. It is not yet clear which position enhances the performance, but it seems as if the placement of the BN layer is problem-dependent. However, recent works have shown that BN performs better if it is placed after the activation function.

2.4.4. Regularization

Regularization is the term used to describe the efforts to reduce the problem of overfitting. CNN networks have an enormous amount of parameters, this means that these networks are prone to overfit. Here, we discuss two types of regularization that are commonly used in practice and used in this work: L1-and-L2 penalties, and dropout.

L1 and L2 penalties apply a penalty to the cost function. Generally in machine learning, the cost function is equal to the loss function, but if we would apply an L1 or L2 penalty the costs of this penalty are added to the loss function. To be more precise, we define a loss function L, and a penalty function P (in this context it represents either an L1 or L2 penalty). These functions all operate on the weights w of the network. We can now describe the cost function denoted as:

$$C(w) = L(w) + P(w). \tag{2.15}$$

Both of the L1 and L2 penalties are configured by one hyper-parameter, which is denoted as λ. If we would apply an L1 penalty, the function P is defined as $P(w) = \lambda * \sum_{x \in w} |x|$. Thus, an L1 penalty is the absolute value of all weights times λ. If we would apply an L2 penalty, the function P is defined as $P(w) = \lambda * \sum_{x \in w} x^2$. Thus, an L2 penalty is the square of all weights times λ. Generally, these regularization norms prevent the network from using gigantic weights, since larger weights imply larger penalties. The effects of L1 and L2 are distinct and depending on the problem either one of these would result in a better classifier.

The last type of regularization we discuss is called dropout. Dropout is a layer that is commonly used in networks that have an enormous amount of parameters, such as deep CNN. Dropout is a deep learning layer specifically designed to address the problem of overfitting [48]. The key idea of dropout is to randomly deactivate activations during the training phase. The probability that an activation is deactivated is configured by the architect of the network but is typically set to 0.5. In [48] the authors show that dropout is more effective than the other discussed regularization methods. However, overfitting still occurs after introducing dropout, a combination of the discussed regularization methods is usually applied.

2.4.5. CNN Architectures

So far we have discussed the layers that comprise CNN architectures, but not the order of the layers. Here, we will discuss the order of the layers that comprise a typical CNN architecture. Typically, a CNN is divided into two parts: a *feature selection block* and a *classification block*. The input of a feature selection block is the

training set, the output of this block is the classification's block input. The feature selection block is comprised of convolutional layers, activations functions (usually ReLU), and pooling.

A feature selection block can be split into various similar blocks, such a block is called a *convolutional block*. The order of the layers in a convolutional block can vary greatly but usually consists of a convolutional layer with ReLU as activation function followed by a pooling operation. Some works vary the times a convolutional layer is applied before the activation functions. Other works, first perform several times a convolutional layer with ReLU as activation function before performing a pooling operation. Such a block only represents a small part of the feature selection block, it is usually extended by the repetition of this block. Additionally, batch normalization is applied if overfitting occurs and is either placer before or after an activation function.

The classification block takes as input the feature selection block's output and performs classification on it. Typically, this block is comprised of a dense layer with ReLU as the activation function. If overfitting is observed dropout layers are added between the dense layers.

2.5. Side-channel Attacks

A side-channel attack (SCA) is an attack that targets the implementation of a system instead of the algorithm. In such an attack, an adversary observes some leakage of information, such as power consumption, electronic radiation, and timing, from a physical channel of the system. The observations are typically called traces in the side-channel community. In an SCA, an adversary aims to deduce the sensitive information from in the system, such as encryption keys, by analyzing the observed traces.

Broadly, SCAs can be split into two categories: *active* and *passive* attacks. In an active attack, the adversary is capable of altering the path of execution, in a passive attack the adversary only observes some leakage. An example of an active attack is a fault injection attack, here the adversary introduces a controlled fault in the system, such that the adversary can exploit the observations to deduce the secret information. A key difference between these attacks is that the victim of the attack can detect an active attack, while this is not true for a passive attack; passive attacks are non-intrusive. In this thesis, we will only discuss passive attacks, but mention active attacks for the sake of completeness. Passive SCAs analyze the observations to extract sensitive information. They can be split into two categories: *profiled* and *non-profiled* attacks. In Figure 2.11 we depict the types of side-channel attacks, we discuss the depicted types in the following sections. However, first, we discuss how the adversary can exploit the information in the traces.

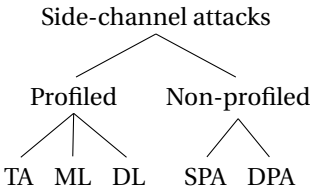


Figure 2.11: Categories of side-channel attacks

SCA exploits the fact that power consumption and EM radiation of a system depends on the processed data. The power consumption of a device can be split into two types, *static* and *dynamic* power. The static power is what the system consumes without any processing. The dynamic power is what the system consumes by processing the data. If a signal transition occurs, this can be observed in a power trace. This is similar for EM radiation, however EM radiation is more precise and has less noise, which is thus the preferred option to perform a side-channel attack.

2.5.1. Leakage Models

Usually, a leakage in traces does no directly leak the sensitive value itself. Therefore it is important to correctly model the leakages from traces, such a model is called a *leakage model*. Common leakage models that are used are *intermediate value* (ID), *Hamming weight* (HW), and *Hamming distance* (HD). The intermediate value leakage model is typically used for software implementations. In this model, the adversary assumes the sensitive value is leaked in the traces. The HW model is usually used for hardware implementations and can be calculated by counting the number of ones from the binary representation of the sensitive value. Here, the assumption is a physical signal transition in the system is visible in the trace. A problem of the HW model is that it suffers from class imbalance, namely, the class four is 70 more likely, than the class zero and eight. In Equation 2.16 we show examples of the leakage models ID and HW for the number 151. HD is typically used

for hardware implementations. It is slightly different from HW, it calculates the XOR operation between two HWs at each bit position.

$$\begin{aligned} ID(151) &= 151 \\ HW(151) = HW(10010111_2) &= 4 \end{aligned} \tag{2.16}$$

2.5.2. Non-profiled Attacks

The first side-channel attack technique to exist is called *Simple Power Analysis* (SPA). In such an attack, the adversary deducts sensitive values by visually analyzing a trace. For example, weak RSA implementations are vulnerable to an SPA attack, in which the adversary deduces the key bits by visually analyzing the amplitudes of a trace. A more powerful attack is called *Differential Power Analysis* (DPA), for which an adversary gathers traces to apply statistical analysis to recover the key [26].

Similar to DPA, *Correlation Power Analysis* (CPA) applies statistical analysis to recover sensitive values. Here, for each possible sensitive value, the adversary correlates it with the traces. The assumption is that if there is a leakage in a feature, the correct guess of the sensitive value correlates significantly more than incorrect guesses of the sensitive value. Since it is unknown which feature leaks information, the adversary correlates each feature with the sensitive values [6].

A possible attack an adversary could perform is a CPA attack. In such an attack, the adversary requires a set of traces of the encryption of arbitrary plaintexts encrypted with the same key and the plaintexts. With these, the adversary calculates for each hypothetical key k the correlation between the traces and the results of $z = \text{S-box}[p \oplus k]$. If the guess of the key k is correct, it would show a correlation between z and the traces. The search space of all possible subkeys is small since these are eight-bit values and thus have 256 possibilities, which causes the attack to be feasible. To retrieve the entire key, the adversary performs such an attack on each subkey.

2.5.3. Profiled Attacks

In a profiled SCA, the adversary has access to two devices: the target device (which is the device under attack), and a copy device (which is identical to the target device). Assuming the devices have similar (or even identical) characteristics it is possible to generate a profile of the copy device to attack the target device. To do so, the adversary gathers a set of traces from the copy device, usually referred to as the profiling traces. Assumed is that the adversary has full control over the copy device, and thus knows the in-and-outputs of operations (including the sensitive values). With the profiling traces and the additional knowledge, a profile of the copy device is made. Subsequently, traces from the target device are gathered, usually referred to as the attack traces. To recover the key, the adversary classifies the attack traces using the built profile. In a profiled attack the success depends on the model's quality, which is dependent on the amount of traces, quality of the traces, and the similarity of the devices.

More formally, we denote the profiling and attack traces as T_p and T_a respectively. Each trace consists of a series of n measurements also referred to as features. A traces from T_p consists of features $x_0, x_1, \ldots, x_{n-1}$. Thus we can see a set of traces as a matrix, where a row represents a trace, and a column a feature from a trace. Additionally to the profiling traces, the adversary knows for each profiling trace the used sensitive value y_p. With this knowledge, the adversary builds a profile f that predicts the sensitive value given a trace. Thus, a profile f maps a trace from domain T to the domain of the sensitive value Y, formally, $f : \mathrm{T} \rightarrow \mathrm{Y}$.

One of the advantages of profiled SCA is a limited amount of traces from the target device are required to recover the key. For example, in [25], the authors only require two traces to recover the key from a protected implementation. However, many traces are required to build a proper model. For the remaining part of this thesis when we refer to an SCA we refer to a profiled SCA unless otherwise specified.

Historically, *Template Attack* (TA) has been used to perform SCAs. It is considered as the most powerful attack from a theoretical perspective [8, 25]. To perform the attack, the adversary builds a multivariate distribution of the profiling traces. Then the adversary uses the distribution and attack traces to find the most likely sensitive value by enumerating all sensitive values. There is however a problem with this approach. Commonly, the traces exist of many features that harden a TA, because a TA requires to calculate the covariance between all features, making it infeasible to perform for many features. Therefore, it is important to select point-of-interests (PoIs) which leak the most. Typically this is done by selecting the features with the most variance.

Other categories of SCAs employ machine learning techniques to learn a model in the profiling phase and predicting the sensitive values in the attack phase. In an ML SCA, the adversary learns a model which predicts

the classes of the sensitive values given the traces. Using this model on the attack traces the adversary is able to predict the sensitive values of the attacks traces. A various range of machine learning techniques have been employed for SCAs such as from random forest and support vector machines [19, 21, 29]. It has been shown that ML performs well in various situations. However, similar to TA, ML performs better when PoIs are selected before performing the attack.

Next to ML, deep learning (DL) techniques have gained attention from the side-channel analysis community. Related works have shown that DL is able to achieve state-of-the-art results, even when countermeasures are present. Most notably, CNNs provide the best performance for a variety of SCA problems.

2.5.4. Metrics

In side-channel analysis, metrics such as accuracy, precision, and recall do not provide a good indication of the performance of an SCA. Instead, in side-channel analysis *guessing entropy* (GE) and *success rate* (SR) are commonly used. GE indicates the average number of key guesses required to recover the key. To calculate the GE, we assume that the adversary outputs the predictions of the probabilities of each key guess: $g = [g_0, g_1, g_2, \ldots, g_{|K-1|}]$, where $|K|$ is the size of the keyspace. Here, for example, g_0 is the prediction of the probability that key as zero is the correct key. The adversary sorts g in descending order, the guessing entropy is then defined as the index or the real key's rank k^* in the sorted probabilities.

Often in SCA, we care about the amount of traces that are required to achieve a guessing entropy of zero; thus the amount of traces required to recover the key, which we will denote as *CGE*. This metric depends heavily on the order the attack traces are processed. If correctly predicted traces are used at the start and incorrectly predicted traces at the end of the computation of the guessing entropy, then the computed guessing entropy provides an inaccurate image of reality. Therefore it is suggested to calculate the partial guessing entropy (*PGE*). To calculate the partial guessing entropy, the adversary runs multiple experiments in which he calculates the GE. For each experiment, the adversary randomly permutes the order of the traces in T_a, and calculates the GE using the new ordered set. The partial guessing entropy is defined as the average of the GE over the experiments. Thus, the partial guessing entropy is the expected value of the guessing entropy. We provide the formal definitions for the *PGE* and *CGE* in the following listing:

Definition 2.5.1. The partial guessing entropy (*PGE*) over n experiments is the average guessing entropy of the experiments. In the experiments the order of the attack traces T_a is randomly permuted.

Definition 2.5.2. The *CGE* is the minimal number of traces required to constantly achieve a *PGE* of zero, such that *PGE* does not change after this point.

Since training a single network could give inaccurate insights into the results, we perform several folds. In each fold, we train a neural network using the same hyperparameters. In this thesis when we refer to the *PGE* and *CGE*, we refer to the average of the performed folds. Consequently, when we report the *CGE* the worst-performing fold decides this metric.

2.5.5. Side-channel Attack on AES

Here we will discuss a side-channel attack on AES. Since AES performs its operations on the state matrix, it is possible to use a divide and conquer approach to recover individual key bytes. Thus, as an adversary, we try to reveal one entry or key-byte at the time. From now on we will refer to key as a key byte of the master key. To perform the attack, an adversary first chooses a leakage model. Here, we will target the first round of AES and the intermediate value. Formally, the leakage model is defined as

$$z = \text{S-box}[k \oplus PT], \tag{2.17}$$

where z is the intermediate value, k the key, and PT the plaintext.

The adversary than learns a model from the profiling set of traces to predict the leakage model. After this, the adversary can use the learned model to generate predictions on unseen traces and attempt to retrieve the key.

To do so, the adversary provides the attack traces T_a to the model, which outputs the probabilities of each intermediate value of the traces. The probabilities are denoted as $C^t = [C_0^t, C_1^t, \ldots, C_{255}^t]$, where C_0^t is the predicted probability for the intermediate value zero and trace t. However, the adversary can not use this to directly retrieve the key. First a mapping between the intermediate value and a key guess has to be made. This is done by enumerating the keyspace, for AES it is in the interval $[0-255]$, and selecting the corresponding

intermediate value. We denote the intermediate value for a key guess $\hat{k}$ and trace t as:

$$z^t = \text{S-box}[\hat{k} \oplus PT^t]. \tag{2.18}$$

The adversary then creates a mapping between the probabilities of C^t and the key guesses, denoted as $P^t = [P_0^t, P_1^t, \ldots, P_{255}^t]$. With the mapping, the adversary assigns the predicted probability of the intermediate value to the corresponding key guess. To prevent floating point issues, the probabilities are converted into log probabilities, and thus summing the probabilities is equal to multiplication and applying the log function. The probability of each key guess $\hat{k}$ is determined as follows:

$$P_{\hat{k}} = \sum_{t \in T_a} P_{\hat{k}}^t. \tag{2.19}$$

P thus contains the summed log probabilities for each key guess. The adversary's best guess is the index with the highest probability of P.

2.5.6. Countermeasures

To decrease the success of a side-channel attack various solutions have been presented, these solutions are called *countermeasures*. Some solutions implement physical countermeasures which make the obtainment of measurements harder, which is called *shielding*. Adversaries can defeat this countermeasure by using more sophisticated equipment and thus provides limited protection.

Dual-rail logic eliminates the ability to obtain power measurements by making the power consumption constant. This ensures that for each bit level operation, the inverse operation is performed as well, and thus the power consumption remains constant. Dual-rail logic is implemented by adding duplicate cell lines that represent the inverse. The disadvantages of this countermeasure are that it does not protect from SCA using EM measurements, the chip's cost and the amount of space on the chip.

Other countermeasures attempt to decorrelate the sensitive value and obtained measurements. Typically, these countermeasures are divided into two categories: *masking* and *hiding*.

A masking countermeasure reduces the leakage of sensitive data by masking the intermediate values. To do so an algorithm is modified such that the intermediate value is masked before an operation is performed with the sensitive value. Thus, an arbitrarily value r is selected to mask the intermediate value x, resulting in the masked value x' shown in Equation 2.20.

$$x' = x \oplus r \tag{2.20}$$

Since x' is not equal to x (unless $r = 0$), the algorithm will work incorrectly. Typical masked implementations fix this by modifying the S-box such that the correct intermediate value is computed depending on r.

Masked implementations can be broken by a *first-order* attack in which two points-of-interest are combined. The two points that are selected should correlate with the mask and key. To protect from such an attack *higher-order* masking has been introduced. In an n-order masking scheme n arbitrary masks are selected and used to mask the data as shown in the following equation:

$$x' = x \oplus r_0 \oplus r_1 \oplus \ldots \oplus r_{n-1}. \tag{2.21}$$

Adversaries can break such an n-order masking scheme by performing an n-order attack. Similarly to the first-order attack, the adversary combines n points that correspond to the masks and key.

Hiding countermeasures produce a randomized timing difference between two identical encryptions. Two well-known types of hiding are *random delay* and *shuffling*. The former type produces a timing difference by performing a random amount of nop operations. The latter does this by shuffling the order of operations. Both types increase the noise in the measurements which makes attacking it harder.

Unfortunately, these countermeasures do no make it impossible to perform SCA but only make it harder to attack. The best protection is obtained by combining a combination of the discussed countermeasures.

2.6. Datasets

In this section, we discuss details about the datasets used for the experiments. The traces are obtained from various AES implementations and range from unprotected to protected. For each dataset, we discuss how the traces are obtained, and the which S-box is targeted.

2.6.1. ASCAD

The ASCAD database is presented in [42]. The database is set up like the MNIST database and has 50 000 profiling traces, and 10 000 attack traces. The traces are recovered from an 8-bit AVR microcontroller (ATmega85515) from a masked implementation of AES-128. The traces were captured from electromagnetic emanation. The database consists of raw traces that contain the measurements from the entire encryption. Next to this, the authors have pre-selected a window in the raw traces that correspond to the S-box operation of subkey three and consists of 700 features. This part of the dataset is used for our experiments. We denote the masked dataset as $ASCAD_M$, and define the leakage function as:

$$Y(k^*) = \text{S-box}[PT_2 \oplus k^*]. \tag{2.22}$$

The unmasked dataset is defined as $ASCAD_U$, and the leakage function is described as

$$Y(k^*) = \text{S-box}[PT_2 \oplus k^*] \oplus M, \tag{2.23}$$

where M is the mask. The dataset can be found `https://github.com/ANSSI-FR/ASCAD`.

2.6.2. DPAv4

DPAcontest v4 exists of 100 000 traces, each consisting of 3 000 features, of a masked AES implementation [44]. However, the traces leak first-order data [36] and this dataset is only used as an unprotected by unmasking the S-box output. We define this dataset as *DPAv4*, and describe the leakage model as

$$Y(k^*) = \text{S-box}[PT_0 \oplus k^*] \oplus M, \tag{2.24}$$

where M is the known mask. This dataset is publicly available at `http://www.dpacontest.org/v4/`

2.6.3. Random Delay

This dataset includes 50 000 traces which each has 3 500 features. The random delay countermeasure presented in [10] is implemented in software on an 8-bit Atmel AVR microcontroller. For this dataset we attack the first key byte. We denote this dataset as *RD* and describe the leakage function as:

$$Y(k^*) = \text{S-box}[PT_0 \oplus k^*]. \tag{2.25}$$

The dataset is publicly available at `https://github.com/ikizhvatov/randomdelays-traces`.

2.6.4. Portability

The traces of this dataset are obtained from a single device by using a near-field EM probe. The device from which the measurements were obtained is an AVR Atmega328p on Arduino Uno, running an unprotected AES-128 implementation. We refer to this dataset as *Porta*. In this dataset, there are 50 000 profiling and attack traces and each trace consists of 500 features. The leakage model we consider for this dataset is shown in Equation 2.26.

$$Y(k^*) = \text{S-box}[PT \oplus k^*]. \tag{2.26}$$

In chapter 6, we discuss more details about this dataset, and it is used for a portability setting.

2.7. Implementation Details

All of the experiments conducted in the following chapters have been performed on the High Performance Computing (HPC) of Delft University of Technology. The machines used for our experiments are all equipped with an Nvidia GTX 1080TI graphic card, which each has 11Gb of memory and 3584 compute cores. The experiments are implemented in python and run with version 3.6.8. For the deep learning experiments, we have utilized the PyTorch framework, specifically version 1.2.0, compiled with CUDA 10.0, using CuDNN cudnn 10.0-7.3.0.29.

3
Related Work

This chapter discusses the related work of this thesis. Here, we will discuss what has been researched in the field of side-channel analysis, and the general direction of these works. Furthermore, we will show why our work is different from the others, and why we think it is an important direction to consider for improving the quality of security assessment of AES implementations.

First, we discuss works that have performed SCAs using machine learning. Subsequently, we discuss literature that have used deep learning techniques for SCAs. As we use deep learning in our work, we discuss some recent advances in deep learning. More specifically, we discuss initialization methods for deep neural networks. After this, we identify where current research is lacking, and denote our research questions.

3.1. Machine Learning SCA

In [18] the authors provide an extensive overview of side-channel attacks conducted by the community. Ever since the attention of the side-channel attack community has shifted from the classical approach to a machine learning approach, the community has experimented with a wide variety of machine learning techniques with great success. The community has conducted comparisons between classical and machine learning, and other various techniques of machine learning. For example, Support Vector Machines (SVMs) have been used to successfully perform an attack on both unprotected and protected implementations [19, 30]. Furthermore, Random Forest (RF) has also been shown to be successful for side-channel attacks [31, 39]. Overall, ML techniques perform well for side-channel attacks, however, in some settings the classical techniques achieve similar performance.

In [13] the authors use a neural network to attack a masked implementation of AES. They use two neural networks for their attack, one which predicts the mask, and one which predicts the key by using the known mask. They show that the neural network approach requires fewer traces to recover the key than the machine learning approach.

Picek et al. [41] discuss the curse of class imbalance and conflicting metrics of ML for side-channel analysis. The class imbalance is present when using the leakage models HW or HD. For these leakage models, some values are way more likely to occur than others. The curse is that ML techniques are affected by the imbalance which causes ML metrics to not be helpful for side-channel analysis. Picek et al. experimentally show that the classical metrics used for machine learning, e.g. accuracy, recall, and precision, can be deceptive when used in a side-channel attack context. Therefore, they suggest focusing on side-channel attack metrics, such as guessing entropy and success rate. This is line with observations in [7] which mentions that accuracy is suited for typical machine learning problems, but not for side-channel analysis.

3.2. Deep Learning SCA

It is unclear which researchers were the first to publish about deep learning for side-channel analysis because many works do not mention the exact network architectures. However, Maghrebi et al. [33] are the first researchers to conduct side-channel attacks using CNNs [33]. In their work, they compare classical machine learning techniques, such as random forest, SVM, with deep learning techniques, such as MLP, CNN, and LSTM. Their experiments show that deep learning has the advantage over classical machine learning techniques, and thus provides better results. They show this for two datasets: one which is an unprotected

implementation, and one which contains a masking countermeasure. Furthermore, their results show that in general CNNs perform well on both datasets.

In [42] the authors introduce a side-channel analysis dataset, also known as the ASCAD database, which has been used in various works of other researchers. Next to the introduction of the dataset, they try to find the best suiting CNN and MLP architecture by analyzing the influence of the hyperparameters. In their work, Prouff et al. [42] show that an increase in the kernel size of a CNN results in better performance when attacking misaligned traces. However, they do not provide a discussion about why increasing the kernel does improve the attack performance. We believe this observation is interesting and required more attention. In chapter 5 we will further analyze the influence of the kernel size.

Both of these works show that CNN performs well in various settings, which is why more research has been conducted on the performance of CNNs. Picek et al. compare the performance of CNNs with machine learning techniques such as Random Forest, XGBoost, and Naive Bayes in [40]. The key aim of their work is to analyze in which settings CNNs provide more performance over the other mentioned techniques. Their experiments show that CNNs provide only improve performance in some settings. They observe that CNNs mainly improve performance when no pre-processing of the traces is done, the level of noise is low, and when the dimensionality of the data is high (i.e. there are many traces with many features). On the other hand, the ML techniques can achieve almost similar performance as CNNs. A key observation is that the ML techniques require significantly less computational power compared to CNNs. Thus, the researchers pose the questions if CNNs are worth it.

Further research on CNNs showed that they can improve the already state-of-the-art solutions for some specific datasets. The datasets have in common that the measurements are obtained from an implementation with a hiding countermeasure. In [7] the authors experimentally showed that CNNs are capable of synchronizing misaligned traces and select the most important features of a trace such that classification can be performed using the selected features. Next to these results, the authors note that this attack is performed with traces where no pre-processing has been done. This is in contrast with a template attack where an adversary would typically realign the traces and select the points-of-interest. The results thus highlight the effectiveness of CNNs on misaligned traces. However, since the used CNN architecture is big and thus complex, it is prone to overfit. Therefore, they introduce two data augmentation techniques for misaligned traces which allows them to generate more training data. They experimentally show the data augmentation techniques work well for misaligned traces.

These findings in [7] are also confirmed by Kim et al. [25], who show that their CNN architecture achieves state-of-the-art performance on the *RD* dataset. Notably, their best network requires fewer attack traces to recover the key for the *RD* dataset than the *DPAv4* dataset, which is regarded as a simple dataset [25]. In their work, the authors experiment with a variety of architectures and datasets. Their experimental results show that no architecture performs well on all the datasets. Therefore, it is important to choose a suitable architecture for the problem at hand. Furthermore, they show that adding noise in the first layer of the network is beneficial for the performance because it reduces overfitting. For small datasets, it is recommended to use a higher level of noise, while for bigger datasets smaller levels of noise provide the best results. In our work, we will use the architectures mentioned in this work as inspiration in chapter 3.

Summarizing these results, CNNs have two important properties that are valuable for side-channel analysis. First of all, they are capable of extracting the most important features without any additional help. Thus pre-processing of the traces is not required to achieve good performance. We consider this a valuable advantage over classical techniques. In [55] the authors mention that pre-processing is error-prone and improper PoI selection results in degraded performance. Secondly, CNNs are spatial invariant, meaning that they can recognize a feature regardless of its position in a feature vector. This property allows CNNs to achieve state-of-the-art performance for datasets that are obtained from implementations with a hiding countermeasure.

The research we have discussed so far used methods that are well-known in the deep learning community. Further research proposed to use more innovative ideas specifically tuned for side-channel attacks which aim to exploit some of its characteristics.

In [17] the authors propose a new CNN architecture that uses additional domain knowledge that is available in side-channel attacks. The knowledge that is added to the neural network architecture is either the plaintext or ciphertext, depending on the leakage model. The domain knowledge is added as an additional feature vector for the classification block of a CNN architecture. In their work, they compare various architectures proposed by different literature and their proposed architecture with and without domain knowledge. They show the architecture which employs the domain knowledge improves performance significantly for unprotected and protected datasets. However, this technique is not suitable if a fixed key is used for the

profiling traces [1].

Zaid et al. [54] highlight the importance of configuring the hyperparameters and architecture; without proper configuration, the models do not perform well. They mention that when we do not comprehend the influence of a hyperparameter we can not achieve the greatest potential of an architecture. To address this problem they introduce three methods used for the explainability and interpretability of each hyperparameter, called weight visualization, gradient visualization, and heatmap. These techniques allow for an adversary to determine the influence of each hyperparameter, which eases the configuration of the hyperparameters. Furthermore, they introduce methodologies for protected and unprotected implementations by using the three visualization methods. A recommendation of their methodology is, specifically meant for datasets with a hiding countermeasure, the kernel size of a CNN should be set to half of the maximum delay of the random delay. This is in contrast with works from the deep learning community from which it is advised to add more layers instead of adding more neurons in a layer [12], also known as the saying 'go deeper, not wider'. Using their proposed methodologies, they developed architectures and experiment with all publicly available datasets, with these they improve the state-of-the-art performance for all datasets. However, as their work achieves state-of-the-art performance for all publicly available datasets, the choice of hyperparameters is sometimes poorly motivated. For example, the authors do not provide a discussion on why different learning rates are used and how these were found for some specific datasets.

The first deep learning layer specifically designed for side-channel attacks, the so-called spread layer, is proposed by Pfeifer and Haddad [38]. With this layer, Pfeifer and Haddad show that fewer layers are required to achieve good results. Additionally, fewer traces are required during the profiling phase, making the learning process faster. These results are interesting for the side-channel analysis community as it means that there is an incentive to develop layers specifically designed to exploit the side-channel characteristics of traces. However, the authors provide minimal motivation on why this layer is able to achieve the reported results, and how to configure the layer's hyperparameters. In chapter 4 we will provide answers to these questions as we will analyze the spread layer in-depth, and improve on some of its flaws.

3.2.1. Network Initialization

Kim et al. [25] have shown that deep CNN architectures perform well for SCAs. However, there are still some problems regarding the training phase of a deep network. The biggest issue is that deep neural networks suffer from vanishing or exploding gradients and are therefore hard to train. In this section, we discuss recent advances in the initialization of deep neural networks that fight these problems.

Before much research was conducted on the initialization of parameters, the parameters were typically drawn from a standard Gaussian distribution. Glorot and Bengio [14] changed this all and proposed a new initialization method, also known as Xavier initialization. This method draws the parameters' values from a Gaussian distribution but takes into account the parameter's amount of inputs and outputs. This method is now the standard parameter initialization for many big deep learning libraries.

As researchers began exploring deep neural network architectures, various works experienced trouble with the convergence of their architectures. For example, the well-known VGG architecture experienced convergence problems and is therefore trained in four stages. In each stage, the network is extended with additional layers and trained to ensure proper convergence [47].

He et al. [16] introduce a new initialization method for deep CNNs. In their paper they discuss that Xavier initialization is designed for linear activations, meaning that this initialization method is not well-suited when ReLU is used. Additionally, they argue that deeper networks have trouble with convergence. To solve these problems they introduce He initialization, specifically designed for CNNs using ReLU, which improves the convergence of deep neural networks over other initialization methods. Another initialization method is proposed in [35], and is called LSUV initialization. This method is not specifically designed for architectures using ReLU as activation function but is a more generic approach suitable for various types of architectures. They verify their proposed method works by experimental validation. Both works have shown that proper initialization of the network's parameters is important for the convergence of deep neural networks.

[1] To explain this, imagine we attack the first round of AES, and thus have as leakage model $z = \text{S-box}[p \oplus k]$. The value of z depends only on the plaintext p if the key k is fixed. Thus when the key is fixed, the plaintext is sufficient to predict z accurately, causing a model to not generalize well to the extent that it overfits massively.

3.3. Portability

Up until recent the published literature only conducted experiments in an unrealistic setting for SCA, namely the profiling and attack traces are gathered from the same device. Additionally, it was not unusual that the same key is used for the profiling and attack traces. Because of this, the results of these studies might paint an incorrect image of the effectiveness of some techniques, such as TA, ML, and DL. Therefore, the SCA community has started to work on a more realistic setting where different devices have been used to obtain the profiling and attack traces.

In recent works it is shown that identical devices have different characteristics, causing the measurements of these devices to have small differences that make realistic SCAs harder [11, 52]. Both works mention that the current research overestimates the attack efficiency of SCAs. It is therefore important to conduct more research in this portability setting.

In [11] the authors propose a method to improve the efficiency of attacks in a portability setting. Their method deals with the difference in the measurements between identical devices by creating a profile using measurements from several copy devices. By doing so they show they can achieve 99% accuracy on the test traces, making the SCA successful.

Around the same time, Bhasin et al. [4] proposed an identical method to deal with the difference between identical devices. In their method, they use traces obtained from multiple identical devices to create the training and validation set. They show that this method achieves better performance than using a single device for training and validation in a portability setting. Furthermore, in their work, they observe that when more traces are used for the profiling phase, the attack efficiency decreases, which could imply that the networks are overspecializing.

3.4. Research Questions

Overall, we are interested in improving the performance of deep learning for side-channel attacks. In the previous sections, we have discussed literature about side-channel analysis using deep learning, but also deep learning itself. Although DL has great potential for SCA, a major problem is the tuning and configuring of hyperparameters. Currently, most of the literature aims at tuning hyperparameters specifically for one dataset. However, recently some interesting works discuss a more generic approach to improve performance. First of all, the work by Pfeifer and Haddad introduces the spread layer designed for side-channel attacks. We are interested in examining if it makes sense to develop a deep learning layer for side-channel attacks for unprotected implementations.

Furthermore, as argued in [25], different architectures perform differently for various datasets. This highlights the importance of picking a suitable architecture and hyperparameters for the problem at hand. To continue on this, as discussed in [42] the authors observe that increasing the kernel size results in increased performance for misaligned traces. However, the kernel size's influence is not the key contribution of their work, and do not discuss the kernel size's influence in depth. Next to the kernel size, [25] shows that deep CNNs can achieve good performance for misaligned traces and decent performance for masked traces. Thus, we see that all the efforts up to now only highlight the best performing architectures, and do not discuss the influence of certain hyperparameters. In this work, we are interested in both the kernel and depth of CNNs for settings in which a countermeasure is present, either random delay or masking. Next to this, we are interested to see if there is a connection between the kernel size and depth of a network. Additionally, by making the distinction between countermeasures we hope to examine if it is possible to generalize the kernel size and depth of an architecture for a specific countermeasure.

As for portability, we see there is still much work required to improve realistic SCAs. We consider a portability setting in which the probe has been moved of position while performing the measurements for the profiling and attack traces. The position change of the probe might result in problems for an SCA. Although this scenario might not seem to be a realistic setting, it is an interesting step towards more realistic settings.

Given these insights, we formulate the following research questions:

RQ 1. *Is there a need for a sepcial designed layer for side-channel attacks on unprotected implementations?*

RQ 2. *What is the influence of the kernel size of CNNs in a side-channel attack where the random delay countermeasure is present?*

RQ 3. *What is the influence of the stacked convolutional layers of a CNN in a side-channel attack context where the random delay countermeasure is present?*

RQ 4. *What is performance tradeoff between the kernel size and stacked convolutional layersof CNNs in a side-channel attack context where the random delay countermeasure is present?*

RQ 5. *What is the influence of the kernel size of CNNs in a side-channel attack where the masking countermeasures is present?*

RQ 6. *What is the influence of the stacked convolutional layers of a CNN in a side-channel attack context where the masking countermeasures is present?*

RQ 7. *What is performance tradeoff between the kernel size and stacked convolutional layersof CNNs in a side-channel attack context where the masking countermeasure is present?*

RQ 8. *Does the repositioning of the probe in-between measurements create a problem for a side-channel attack? And if so, what can we do to fight this problem?*

We present the answers to the research questions in the following chapters. More specifically, **RQ 1.** is answered in chapter 4, **RQ 2.** till **RQ 7.** are answered in chapter 5, and **RQ 8.** is answered in chapter 6. Additionally, we explicitly answer all the research questions in chapter 7.

4

The Need of a Custom Deep Learning Layer for SCAs

In this chapter, we analyze if there is a need for a specially designed deep learning layer for unprotected AES implementations. The idea for such a layer has been introduced by Pfeifer and Haddad in [38]. They introduce a new deep learning layer called the *spread* layer. In their work, they compare two network architectures, an architecture which utilizes the spread layer, and an architecture presented in [42]. They show that the architecture which utilizes the spread layer requires fewer attack traces to recover the key. The most remarkable result of their work is that fewer traces are required during the profiling phase in comparison to other network architectures. Their results are promising since it could imply that it makes sense to develop a deep learning layer that can exploit the information in a trace specific to a first-order SCA.

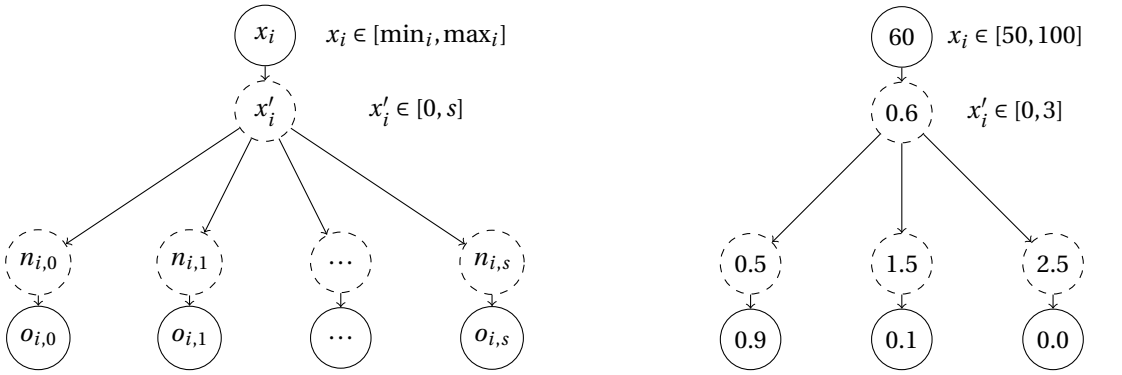
However, the authors provide limited argumentation on why an architecture with a spread layer is efficient for unprotected implementations. Arguing why there is a performance difference between these network architectures could lead to more meaningful insights that could improve a profiled SCA even more. Another limiting factor of their work is that the authors only compare the performance of their presented architecture with the architecture presented in [42] on the ASCAD dataset. These architectures are not quite similar and make it hard to make a fair comparison on the performance.

To analyze if there is a need for a deep learning layer for SCAs on unprotected implementations we first analyze the spread layer. Therefore we first provide a technical discussion of the spread layer in section 4.1. It should be mentioned that we encountered an error in the explanation of the spread layer while reading the original paper. We have communicated this with the authors, who resolved the error in an updated version of the paper. After the technical discussion, section 4.2 discusses the architectures that have been utilized to compare architectures that employ the spread layer and which do not. In section 4.3, the resulting guessing entropy of the networks for various datasets is shown and discussed. The results presented in this chapter differ significantly from the results presented by Pfeifer and Haddad.

Therefore, we analyze the influence of the spread factor, the spread layer's only hyperparameter. Surprisingly, we show that the spread factor's influence is minimal, and to explain these result we provide an analysis of the internal behavior of the spread layer in subsection 4.3.2. From this analysis, it is clear that the presented spread layer by Pfeifer and Haddad does not operate as intended. We propose two modifications of the spread layer, denoted as $Spread_{L2}$ and $Spread_{L3}$. We conduct experiments with the two layers and discuss their attack efficiency in comparison with the already known layers. These comparisons provide us with insights into the question if there is a need for a specially designed deep learning layer for SCAs. In section 4.4 we conduct and discuss results of the spread layer and improved versions on protected implementations. In the following section, section 4.5, we conduct experiments on protected implementations of AES and discuss the results. Finally, we provide a conclusion of the presented results in section 4.6.

4.1. The Spread Layer

The spread layer's objective is to efficiently distinguish between all possible Hamming weights [38]. To achieve this, the spread layer transforms its input into a spatially encoded output. This causes a dimension increase which can be tuned by the hyperparameter called the spread factor, which is denoted as s. To explain how



(a) The inner workings of the spread layer for an arbitrary spread factor s.

(b) An example of the inner workings of the spread layer where the spread factor $s = 3$.

Figure 4.1: Examples of the inner workings of the spread layer. The left figure depicts the structure and information flow, while the right figure depicts an example with real values.

the spread layer works, we discuss the internals for a single neuron.

We refer to the input vector of the spread layer as x, and x_i as the ith entry in this vector, which thus represents the ith neuron. For each entry of the input vector, there are s output neurons. We refer to the jth output neuron of an entry x_i as $o_{i,j}$, note that $0 \leq j < s$. Furthermore, each output neuron has a so-called centroid $n_{i,j}$ assigned, for which its value is calculated by the formula:

$$n_{i,j} = 0.5 + j. \tag{4.1}$$

The centroids are used to determine which output neurons are used. To calculate the output of a neuron x_i the spread layer linearly maps x_i to x'_i in the interval $[0, s]$. To achieve this, the spread layer keeps track of the minimum and maximum values of x_i seen over the entire dataset during the training phase. We refer the minimum and maximum of x_i as $\min_i$ and $\max_i$ respectively, during the training phase. The value of x'_i is calculated by the function m:

$$x'_i = m(x_i, \min_i, \max_i) = \frac{x_i - \min_i}{\max_i - \min_i} * s. \tag{4.2}$$

Then, the value of $o_{i,j}$ is calculated by the function f as shown in Equation 4.3.

$$o_{i,j} = f(x'_i, n_{i,j}) = \begin{cases} 1 & \text{if } n_{i,j} = 0.5 \wedge x'_i \leq n_{i,j} \\ 1 & \text{if } n_{i,j} = s - 0.5 \wedge x'_i > n_{i,j} \\ max(0, 1 - abs(n_{i,j} - x'_i)) & \text{otherwise} \end{cases} \tag{4.3}$$

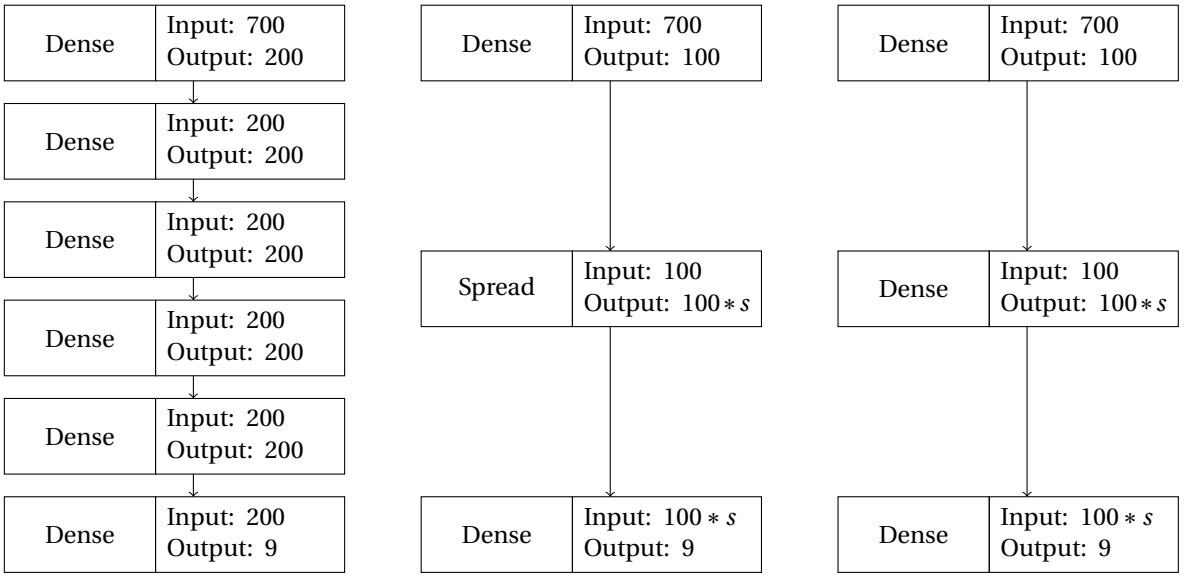
The function f outputs values in the interval $[0, 1]$. For a value x_i and the remapped value x'_i, if x'_i is equal to one of the corresponding centroids $n_{i,j}$ then $o_{i,j}$ is the only neuron with a non-zero output. Otherwise, there are at most two neurons with a non-zero output, which are the ones for which x'_i is the closets to the corresponding $n_{i,j}$. Furthermore, the output neurons $o_{i,j}$ always sums up to one: $\sum_{0 \leq j < s}^{j} f(x'_i, n_{i,j}) = 1$. Using the functions from m and f we calculate $o_{i,h}$ for neuron x_i with the function g:

$$g(x_i, \min_i, \max_i, n_{i,j}) = f(m(x_i, \min_i, \max_i), n_{i,j}). \tag{4.4}$$

In Figure 4.1 we depict an example of the inner workings of the spread layer plus a toy example with real numbers. In these figures, the dashed circles represent internal values while the non-dashed circles represent the values of the in-and-output neurons. Figure 4.1a shows an example when the spread factor is equal to s. This figure shows the remapping of x_i to x'_i, followed by the calculation of each output neuron, $o_{i,0}, o_{i,1}, \ldots, o_{i,s}$. Figure 4.1b shows an example with numbers, here $s = 6$, $x_i = 60$, $\min_i = 50$ and $\max_i = 100$. By applying the function g three times, once for each output, we calculate the value of the neurons: $o_{i,0} = g(60, 50, 100, 0.5) = 0.9$, $o_{i,1} = 0.1$ and $o_{i,2} = 0.0$.

4.2. Architectures

The results of the first-order attack in [38] are impressive, however in their work they merely make a comparison between their presented architecture, $\boldsymbol{Spread_{PH}}$, and the architecture $\boldsymbol{MLP_{best}}$, presented in [42]. The



(a) The architecture of $\boldsymbol{MLP_{best}}$ (b) The architecture of $\boldsymbol{Spread_{PH}}$ (c) The architecture of $\boldsymbol{MLP_{RT}}$

Figure 4.2: The architectures of $\boldsymbol{MLP_{best}}$, $\boldsymbol{Spread_{PH}}$ and $\boldsymbol{MLP_{RT}}$. The architectures $\boldsymbol{Spread_{PH}}$ and $\boldsymbol{MLP_{RT}}$ use the spread factor s to determine the amount of neurons in the second hidden layer.

hyperparameters of the latter architecture were specifically tuned to the dataset which Pfeifer and Haddad have used to analyze their architecture. Therefore, it is to be expected that $\boldsymbol{MLP_{best}}$ will perform well against this dataset. The results in [38] show that $\boldsymbol{Spread_{PH}}$ significantly outperforms $\boldsymbol{MLP_{best}}$, which thus highlights the remarkable results of the $\boldsymbol{Spread_{PH}}$ architecture.

However, the two architectures are not similar, for example, $\boldsymbol{Spread_{PH}}$ has three hidden layers, while $\boldsymbol{MLP_{best}}$ has six. Furthermore, the number of neurons of the models differ vastly, which also implies that the capabilities of the models could differ vastly. The results presented by Pfeifer and Haddad could thus be explained by the fact that $\boldsymbol{Spread_{PH}}$ is the superior model for the $ASCAD_U$ dataset, and thus not because $\boldsymbol{Spread_{PH}}$ utilizes a spread layer. To analyze if the spread layer could improve the performance, a comparison with a similar architecture should be made.

Here, we introduce this similar architecture, which will be referred to as $\boldsymbol{MLP_{RT}}$. $\boldsymbol{MLP_{RT}}$ has an equal amount of layers and neurons in each layer as $\boldsymbol{Spread_{PH}}$. In Figure 4.2 we depict a visualization of the architectures $\boldsymbol{MLP_{best}}$, $\boldsymbol{Spread_{PH}}$, and $\boldsymbol{MLP_{RT}}$. The key difference between $\boldsymbol{Spread_{PH}}$ and $\boldsymbol{MLP_{RT}}$ is that $\boldsymbol{Spread_{PH}}$ utilizes a spread layer while $\boldsymbol{MLP_{RT}}$ a dense layer. Furthermore, each mentioned architecture utilizes the ReLU activation function in each layer, except for the last layer which uses the softmax activation function.

4.3. Attacks on Unprotected Implementations

Since the source code of the spread layer is not publicly available, we have implemented the layer in PyTorch [1]. This section is outlined as follows. First, to verify if our implementation is correct we attempt to reproduce the results as presented by Pfeifer and Haddad and conduct attacks on the dataset $ASCAD_U$. In the next section, we analyze the spread factor's influence. After this analysis, a discussion of the spread layer's activations is given. At last, we discuss the most remarkable results from all conducted experiments.

4.3.1. Reproducibility

The most remarkable result presented in [38] is that $\boldsymbol{Spread_{PH}}$, while trained with 1 000 traces of $ASCAD_U$, can retrieve the key in around 1 725 traces , while $\boldsymbol{MLP_{RT}}$ achieves only a guessing entropy of 7.7 after 5 000 traces. This section attempts to reproduce these results by using similar settings as presented in [38], which are listed in Table 4.1. Next to reproducing these results, the three mentioned architectures have been trained using the same set of traces, such that a comparison between these architectures can be made.

In Figure 4.3 we depict guessing entropy plots of the trained classifiers. The results shown in this figure

[1] For more details about the used versions, we refer to section 2.7.

Batch size	Learning rate	Spread factor	Epochs	Training size	Leakage model
100	1e−4	6	80	1 000	HW

Table 4.1: Hyper parameters used to reproduce the results of the first order attack on the ASCAD database presented by [38]

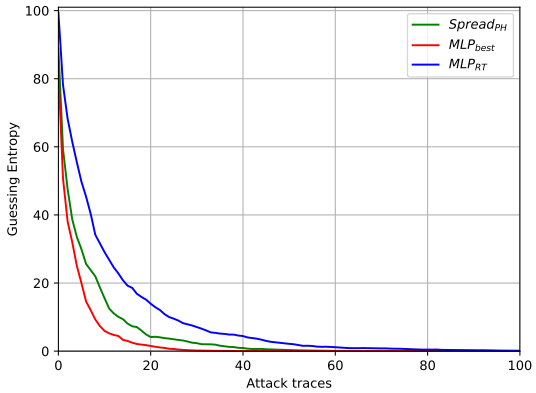


Figure 4.3: A plot of the guessing entropy of the three architectures $\boldsymbol{Spread_{PH}}$, $\boldsymbol{MLP_{best}}$, and $\boldsymbol{MLP_{RT}}$ trained with 1 000 traces from $ASCAD_U$, and Hamming weight as leakage model.

are significantly different from the results presented in the work by Pfeifer and Haddad. Here, all the classifiers perform well in terms of guessing entropy and retrieve the key in less than 100 traces. In the work by Pfeifer and Haddad, $\boldsymbol{Spread_{PH}}$ requires 1 725 traces to retrieve the key, while $\boldsymbol{MLP_{best}}$ fails to retrieve the key. Furthermore, this figure shows that $\boldsymbol{MLP_{best}}$ performs the best, then $\boldsymbol{Spread_{PH}}$, and finally $\boldsymbol{MLP_{RT}}$. The difference between $\boldsymbol{Spread_{PH}}$ and $\boldsymbol{MLP_{RT}}$ is minimal, but the network $\boldsymbol{Spread_{PH}}$ requires twice as few traces to recover the key.

The difference between the presented results in this work and by Pfeifer and Haddad are hard to explain. Their results show that $\boldsymbol{MLP_{best}}$ can not retrieve the key when trained on 1 000 traces. However, in [42] a first-order attack using $\boldsymbol{MLP_{best}}$ can retrieve the key in four traces. Therefore, it seems more than natural that the results shown in this section are more compelling than the presented results by Pfeifer and Haddad, and the authors seem to have made a mistake.

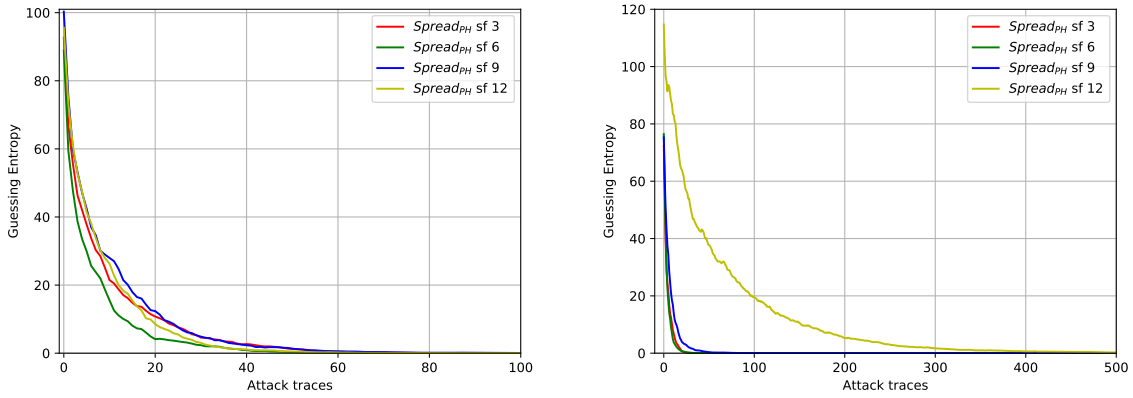
4.3.2. Varying Spread Factors

To analyze if $\boldsymbol{Spread_{PH}}$ has a benefit over the other $\boldsymbol{MLP_{best}}$ and $\boldsymbol{MLP_{RT}}$, we have performed various experiments with a wide variety of hyperparameters. In Table 4.2 we list the hyperparameters used for the experiments, in short, we vary the learning rate, batch size, spread factor, leakage model, and the training size. Furthermore, we will use the datasets $ASCAD_U$ and *DPAv4* for the experiments. We vary the spread factor because it is the only hyperparameter of the spread layer and thus can have an influence on the performance of a network.

Learning rate	Batch size	Spread factor	Train size	Leakage model
{1e−2, 1e−3, 1e−4}	{100, 200}	{3, 6, 9, 12}	{1 000, 5 000, 20 000, 40 000}	{HW, ID}

Table 4.2: Hyperparameters configurations for all folds

When the networks are trained with more than 5 000 traces, we observe no difference between the results. In general, this holds as well when the architectures are trained with less than 5 000 traces but for some exception. The experimental results for the *DPAv4* trained with 1 000 traces show the spread factors 12 performs significantly worse than the other spread factors. In Figure 4.4 we depict the experimental results for $ASCAD_U$ and *DPAv4*, trained with the hyperparameters listed in Table 4.3. These results highlight that $\boldsymbol{Spread_{PH}}$, in general, performs similarly for various spread factors. This is remarkable since we expected that the spread factor's influence would have a noticeable effect on the classifier, and thus the guessing entropy.



(a) $ASCAD_U$ (b) *DPAv4*

Figure 4.4: The results of using the ***Spread*$_{PH}$** trained on 1 000 traces with spread factors 3, 6, 9, and 12. The left and right figures show the guessing entropy using the datasets $ASCAD_U$ and *DPAv4* respectively. Both of these results are obtained using the Hamming weight as leakage model.

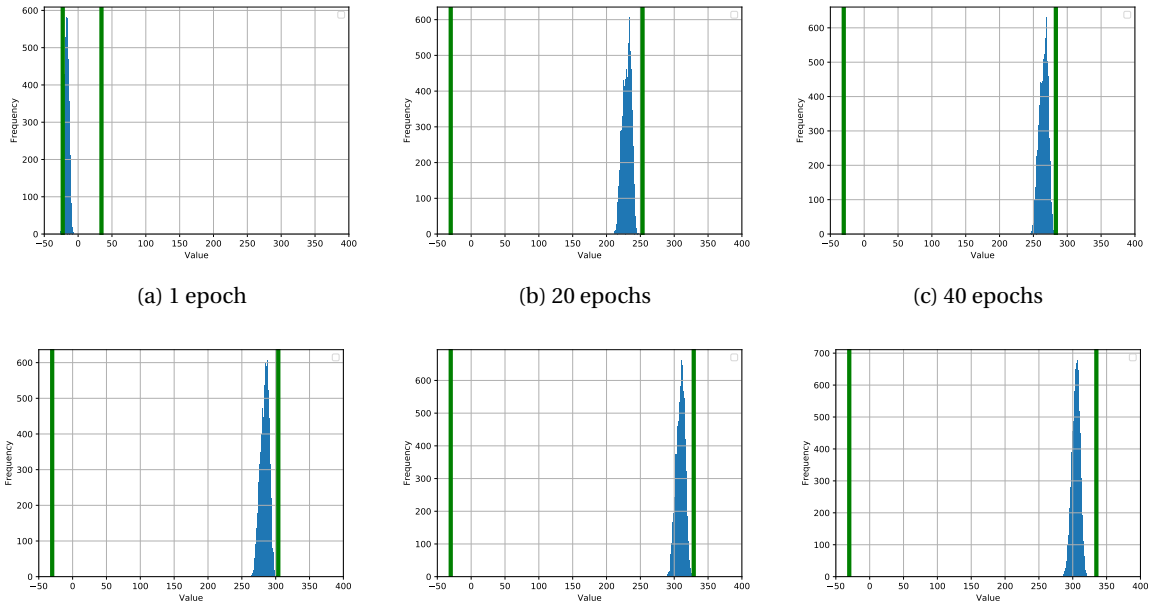
Batch size	Learning rate	Spread factor	Epochs	Training size	Leakage model
100	1e−4	{3, 6, 9, 12}	80	1 000	HW

Table 4.3: Hyperparameters used for the results shown in Figure 4.4.

Since we observe no significant difference in guessing entropy between a high or low spread factor, we analyze the spread layer's activations, thus we analyze the spread layer's output. To do so, we have trained ***Spread*$_{PH}$** with traces from $ASCAD_U$ and the hyperparameters listed in Table 4.3, but with a spread factor of 6. We gather the activations by supplying the entire attack set (10 000 traces) to the network and storing the spread layer's activations. An analysis of the stored activations showed that around 58% is equal to zero. Thus, for the entire attack dataset, the spread layer fails to spatially spread its input.

This result implies that the spread layer's input values are mapped to the same centroid, and thus perform the remapping of its input (in the interval $[0, s]$) incorrectly. Recall, that $\min_i$ and $\max_i$ for a neuron x_i are learned by selecting the minimum and maximum observed during the training phase. This causes problems during learning because the distribution of x_i shifts, and as a result, $\min_i$ or $\max_i$ reflects an incorrect image of the true minimum and maximum of a spread layer's neuron x_i. The shift occurs because we perform backpropagation during the training phase, causing the weights and biases of the layer prior to the spread layer to be adjusted. As the network learns an incorrect minimum or maximum, the spread layer maps its input always to the same output neuron. In other words, the spread layer does not account for the shift of a neuron's distribution caused by backward propagation and thereby loses its power to spatially encode the data.

To confirm this theory, we train ***Spread*$_{PH}$** with the same hyperparameters as in the previous paragraph, but for 100 epochs and every 20 epochs the network is stored, such that we can observe the evolution of the neuron's distributions over several epochs. We analyze this evolution by visualizing the $\min_i$, $\max_i$, and the neurons' distribution for each stored network. In Figure 4.5 we depict the evolution of a single neuron's distribution after one epoch, 20, 40, 60, 80, and 100 epochs. In these figures, the green lines represent $\min_i$ and $\max_i$, while the blue lines represent the distribution. From these figures, it is clear that after more epochs (and thus more backward passes), $\min_i$ and $\max_i$ are far from equal to the true minimum and maximum. Since this figure only depicts the results for a single neuron, we can not generalize this for all neurons. To confirm the shift in distribution occurs at all neurons of the spread layer, the mean squared error (MSE) between the true minimum and maximum, and $\min_i$ and $\max_i$ are calculated respectively. Table 4.4 lists the MSE after each epoch. This table shows that as a model is trained more, the MSE grows, and thus $\min_i$ and $\max_i$ drift away from the true minimum and maximum respectively.



(a) 1 epoch (b) 20 epochs (c) 40 epochs

(d) 60 epochs (e) 80 epochs (f) 100 epochs

Figure 4.5: Intermediate values of a single neuron of the input of the spread after different epochs. The green lines represent the minimum and maximum learned by the network. The blue boxes represent a histogram and reveal the distribution of the values. This figure shows that the distribution of the values shift, while the learned minimum remains the same.

Epochs	MSE minimum	MSE maximum
1	977.03	1 470.50
20	10 577.30	12 423.04
40	12 937.62	19 314.32
60	15 430.71	25 413.44
80	19 045.78	30 516.95
100	20 846.99	37 361.19

Table 4.4: Mean squared errors between $\min_i$ and the true minimum, and $\max_i$ and the true maximum.

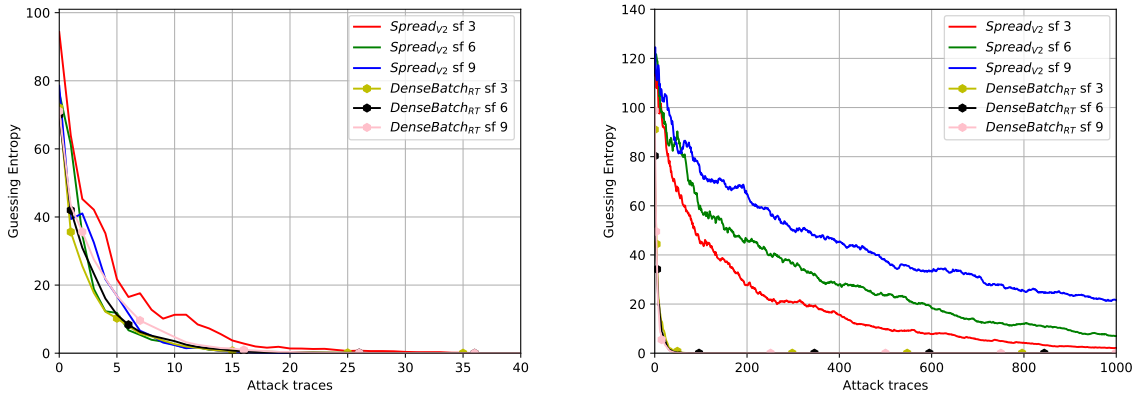
Spread Version 2

To overcome the issue that $\min_i$ or $\max_i$ drifts away from the true minimum or maximum, the manner in which these values are learned should be modified. We do this by adding a batch normalization layer before the spread layer. Recall, that a batch normalization layer first transforms its input to the standard normal distribution (referred to as $\hat{x}_i$), and then transform it to $x_i = \hat{x}_i \cdot \gamma + \beta$, where γ and β are learnable parameters. Since x_i is normally distributed we can determine $\min_i$ and $\max_i$ such that $\min_i \leq x_i \leq \max_i$ with a negligible probability. For a standard normal distribution Z, we know that $P(Z < -4) \approx 0.0$, and thus $P(Z > 4) \approx 4$. The following equation shows how to calculate the minimum and maximum.

$$\begin{aligned} \min_i &= -4.0\gamma + \beta \\ \max_i &= 4.0\gamma + \beta \end{aligned} \tag{4.5}$$

By using these formulas we are certain that $\min_i$ and $\max_i$ are close approximations of the true minimum and maximum. We have implemented these changes to the spread layer and will refer to this layer as $Spread_{L2}$ (note that this layer also includes the batch normalization layer).

To analyze if $Spread_{L2}$ suffers from the same weaknesses as the spread layer we perform the same experiment with $\boldsymbol{Spread_{V2}}$ as we performed on $\boldsymbol{Spread_{PH}}$ in the previous section. $\boldsymbol{Spread_{V2}}$ is similar to $\boldsymbol{Spread_{PH}}$, however, the spread layer is replaced by $Spread_{L2}$. The experimental results show that around 40% of the output neurons of the spread layer are not active (i.e. have a zero output). As expected, this is less than $\boldsymbol{Spread_{PH}}$, however, more than one-third of the neurons are not active and could thus be considered as useless. In the



(a) Hamming weight as leakage model

(b) Intermediate value as leakage model

Figure 4.6: Guessing entropy plot of $\boldsymbol{Spread_{V2}}$ and $\boldsymbol{Dense_{BN}}$ trained with 1 000 from $ASCAD_U$. The left figure depicts the results with HW as leakage model while the right image shows the results for intermediate value as leakage model.

following section, we further analyze why this behavior occurs, but first, we compare the guessing entropy of $\boldsymbol{Spread_{V2}}$ with a similar architecture.

Since $\boldsymbol{Spread_{V2}}$ utilizes a batch normalization layer while the previously introduced architectures do not, comparing these is not fair. Therefore we introduce an additional architecture that contains a batch normalization layer, which we denote as $\boldsymbol{Dense_{BN}}$. This architecture is similar to $\boldsymbol{MLP_{RT}}$, but instead of a dense layer, a batch normalization layer is used at the second hidden layer. We will first show plots of the guessing entropy and then analyze the spread layer's activations.

In Figure 4.6 we depict the experimental results of $\boldsymbol{Spread_{V2}}$ and $\boldsymbol{Dense_{BN}}$ trained with 1 000 traces of $ASCAD_U$ for the leakage models HW and intermediate value. Note that for both leakage functions $\boldsymbol{Spread_{V2}}$ outperformed $\boldsymbol{Spread_{PH}}$. We believe this is the result of the batch normalization layer in $\boldsymbol{Spread_{V2}}$ since $\boldsymbol{Dense_{BN}}$ also performs well. Figure 4.6a depicts the results for which HW is used as a leakage function. Here we observe no significant difference in guessing entropy between the two architectures. This figure also shows that different spread factors have little to no influence on the guessing entropy. In Figure 4.6b we depict the results for intermediate value as leakage model. We observe that $\boldsymbol{Dense_{BN}}$ outperforms $\boldsymbol{Spread_{V2}}$ for all experimented spread factors. This figure also highlights the difference in guessing entropy for the spread factors. It highlights that a lower spread factor provides better performance when intermediate value is used as leakage model. We believe a lower spread factor, when trained with a little amount of traces, provides better performance because the spread layer has fewer output neurons. If the spread factor is higher there are more neurons, and thus more parameters which require to be trained as well, which is hardened when there is a limited amount of training traces. Other architectures that do not utilize a spread layer do not suffer from this, because the spread layer only uses at most two of the possible output neurons. Thus, increasing the spread factor yields in output neurons that are less trained than other neurons, especially when they are situated at the borders. We confirm this theory by increasing the training size to 40 000 traces, by doing so we perform more backwards passes. In Figure 4.7 we depict the results of $\boldsymbol{Spread_{V2}}$ and $\boldsymbol{Dense_{BN}}$ trained with 40 000 traces of $ASCAD_U$, and intermediate value as leakage model. From this figure, we observe that the architectures perform similarly and thus the theory seems to be correct.

As we have analyzed the spread layer's activations for $\boldsymbol{Spread_{PH}}$, we also analyze the activations of $Spread_{L2}$ for $\boldsymbol{Spread_{V2}}$. We perform the same experiment as for $\boldsymbol{Spread_{PH}}$, the results show that around 39% of the neurons are not active (i.e. equal to zero). This is an improvement over the 58% of the spread layer, however, this is not what we expected or intended. Therefore, we have performed the same analysis as for $\boldsymbol{Spread_{PH}}$, such that we can confirm if $\min_i$ and $\max_i$ are set correctly for all inputs x_i. This analysis showed that these values were set correctly, thus another problem is present.

Further analysis showed that for a neuron x_i, the output neurons $o_{i,0}$ and $o_{i,5}$ were barely used. This is caused by how the spread layer determines which output neuron is used, and the input distribution. Recall that spread determines which output neurons should be used by comparing the input value with the centroids. Additionally, the centroids are calculated by a linear function. Since the input of the spread layer is

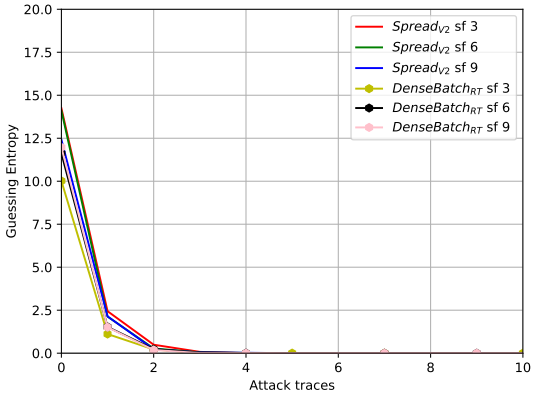


Figure 4.7: Guessing entropy plot of ***Spread$_{V2}$*** and ***Dense$_{BN}$*** trained with 40 000 from $ASCAD_U$ using intermediate value as leakage model.

normally distributed the output neurons which are situated at the border are rarely used. Thus the spread layers' input is expected to be uniformly distributed while the input is normally distributed.

Spread Version 3

To fix the problem of the difference between the expected and actual distribution of the spread layers' input, we suggest to modify the spread layer such that it expects a normally distributed input. In this section, we have implemented this solution and analyzed the sparsity of the layer's output activations. Additionally, we compared the performance of the layer with a similar architecture. Before we discuss these results we explain how the newly proposed layer, called $Spread_{L3}$, is implemented.

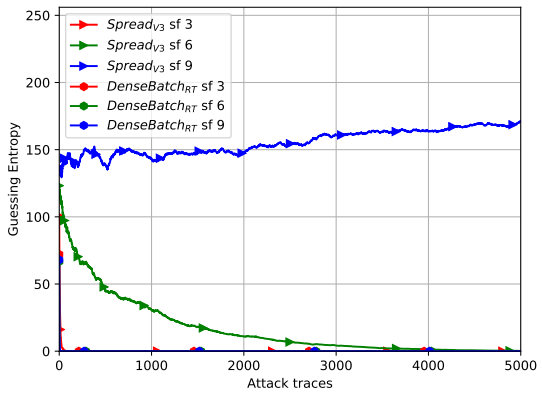
The difference between $Spread_{L2}$ and $Spread_{L3}$ is that centroids' values are calculated differently (Equation 4.1 shows the formula to calculate the centroids for the spread layer). Since the spread factor is known, and $Spread_{L2}$ its input is drawn from the distribution $\alpha \cdot \mathcal{N}(0,1) + \beta$, it is possible to select centroids such that each output neuron has the same probability to be used by the network. First, we calculate the centroids' values when α and β are equal to zero, using the inverse of the cumulative probability density function $\mathcal{N}$. To be precise,the unscaled value of a centroid n'_i is calculated such that $P(\mathcal{N} \leq n'_i) = y_i$, where $y_i = \frac{1}{s} \cdot (i+1)$, $i \in \{0, ..., s-1\}$, and s the spread factor. Then the value of the unscaled centroid is scaled such that we retrieve the value of the centroid n_i as follows:

$$n_i = \alpha \cdot n'_i + \beta. \tag{4.6}$$

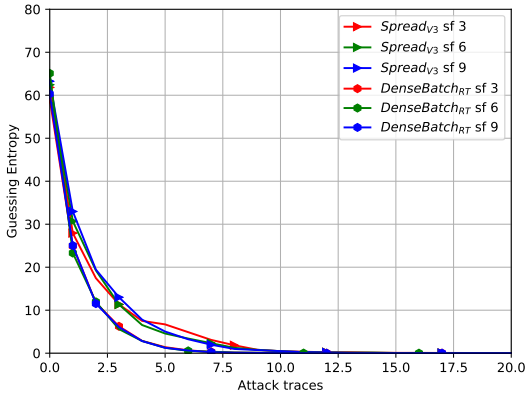
We denote the architecture which employs $Spread_{L3}$ as ***Spread$_{V3}$***. The only difference between ***Spread$_{V2}$*** and ***Spread$_{V3}$*** is that $Spread_{L2}$ has been interchanged with $Spread_{L3}$.

We have performed the same experiments as we have done for ***Spread$_{PH}$*** and ***Spread$_{V2}$***, and observed that all output neurons of $Spread_{L3}$ are non-zero. Thus the changes to the spread layer work as intended. Therefore, we compare the performance of ***Spread$_{V3}$*** with ***Dense$_{BN}$***. Like the previous experiments, we have experimented with the spread factor, training size, and leakage model. In Figure 4.8 we depict the guessing entropy of ***Spread$_{V3}$*** and ***Dense$_{BN}$***. We will discuss the results of ***Spread$_{V3}$*** with various spread factors and subsequently compare the two architectures. First, we discuss the results when the networks were trained with HW as leakage model, followed by the results of the networks trained with intermediate value as leakage model.

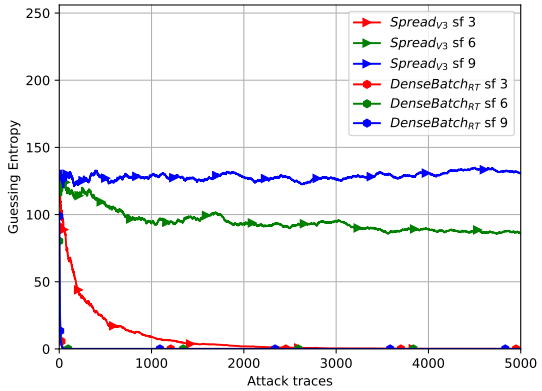
If the networks are trained with 1 000 traces and HW as leakage model we observe that a low spread factor yields in the best guessing entropy. When the spread factor is equal to 9 we observe that ***Spread$_{V3}$*** can not retrieve the key, while the other spread factors can recover the key. The difference in *CGE* is however significant when the spread factor is equal to 3 we require around 200 times more traces than when the spread factor is 6 to successfully attack this dataset. We believe that a higher spread factor results in worse performance because of similar reasons what we observed in the previous section. A higher spread factor and limited amount traces means there are more neurons that require backpropagation to adjust its weights. As the spread factor increases, there is an increase in the spread layer's output activations which are equal to zero. Combining this with a limited amount of traces, and thus a limited amount of backward propagation, the parameters of these neurons are less adjusted to fit over the training data. When ***Spread$_{V3}$*** is trained with 40 000 traces we



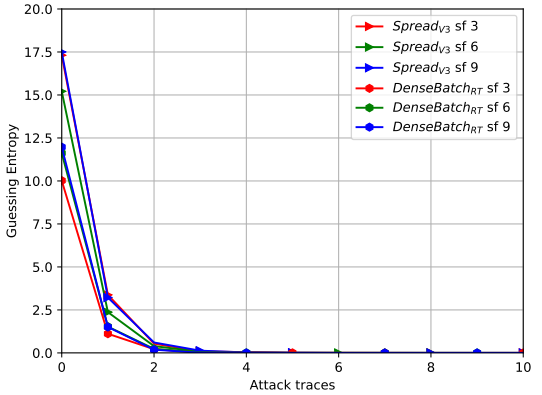
(a) Training size 1 000 and Hamming weight as leakage model



(b) Training size 40 000 and Hamming weight as leakage model



(c) Training size 1 000 and intermediate value as leakage model



(d) Training size 40 000 and intermediate value as leakage model

Figure 4.8: The guessing entropy of ***Spread$_{V3}$*** and ***Dense$_{BN}$*** trained with 1 000 and 40 000 traces of *ASCAD$_U$* using HW and intermediate value as leakage model.

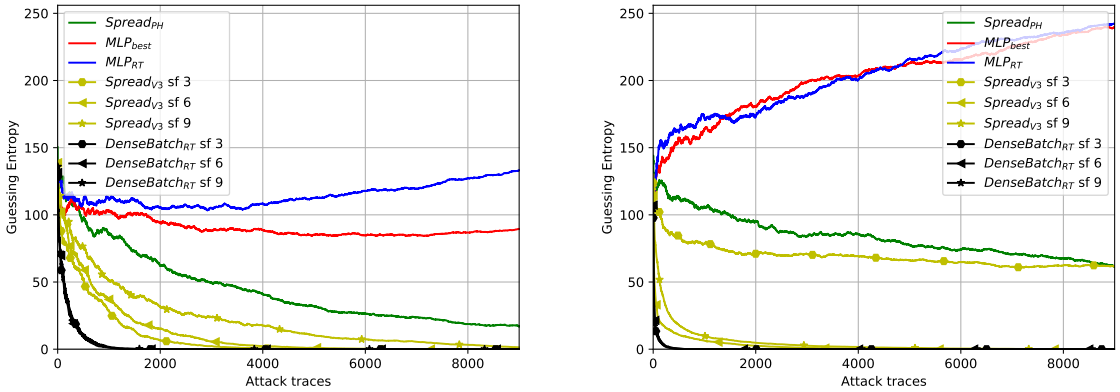
see no difference in guessing entropy between the spread factors. This is an indication that our theory, on why higher spread factors result in decreased performance, is correct.

In Figure 4.8c and Figure 4.8d we depict the results when intermediate value is used as leakage model. The results are similar to the results of HW. The major difference is that the networks trained with 1 000 traces are significantly worse in performance than when hamming weight is used as leakage model. However, this is the opposite when 40 000 training traces are used; then intermediate value outperforms HW.

The difference in the performance of ***Spread$_{V3}$*** and ***Dense$_{BN}$*** trained with 1 000 traces for both leakage models is significant. However, when the architectures are trained using 40 000 traces we see little to no difference. For ***Dense$_{BN}$*** there is no significant difference in performance when trained on less or more traces. Therefore we see no reason to use *Spread$_{L3}$* in a deep learning architecture.

4.4. Attacks on Protected Implementations

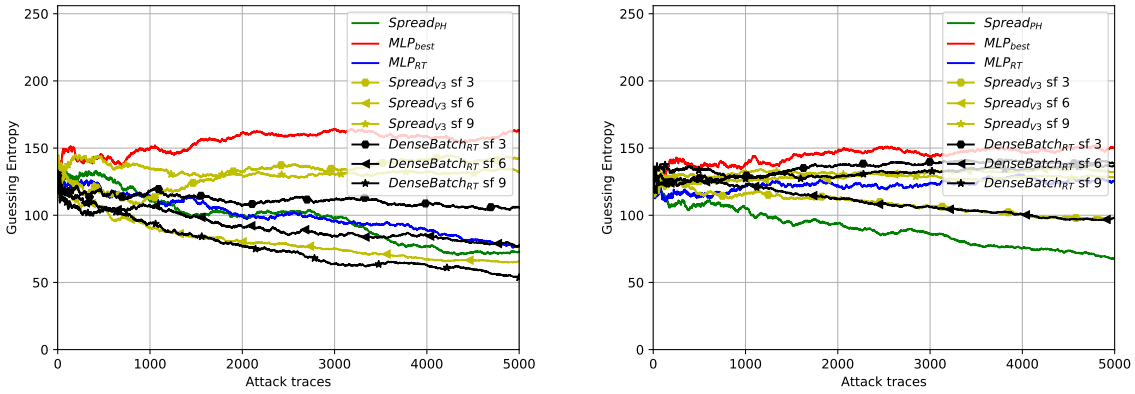
So far we have experimented with datasets that were obtained from unprotected AES implementations, here in this section we perform attacks on protected AES implementations and discuss the obtained results. Like in the previous section we will use ***Spread$_{PH}$***, ***MLP$_{best}$***, and ***MLP$_{RT}$*** for our experiments. Next to these architectures, we use the proposed improvement of the spread layer, *Spread$_{L3}$*, utilized in ***Spread$_{V3}$***, and ***Dense$_{BN}$*** for a fair comparison. For our experiments, we will use the datasets *ASCAD$_M$* and *RD*. Furthermore, the hyperparameters used are the same as presented in subsection 4.3.2. Since we attack protected AES implementations we do not expect to retrieve the key. However, we are still interested in the performance of the architectures



(a) Hamming weight as leakage model

(b) Intermediate value as leakage model

Figure 4.9: Plot of the guessing entropy of the architectures $\boldsymbol{Spread_{PH}}$, $\boldsymbol{MLP_{RT}}$, $\boldsymbol{MLP_{best}}$, $\boldsymbol{Spread_{V3}}$ and $\boldsymbol{Dense_{BN}}$ trained with 40 000 training traces from $ASCAD_M$ using HW and intermediate value as leakage model.



(a) RD trained on 40 000 traces hamming weight

(b) RD trained on 40 000 traces leakage: id

Figure 4.10: Plots of the resulting guessing entropy of the architectures $\boldsymbol{Spread_{PH}}$, $\boldsymbol{MLP_{RT}}$, $\boldsymbol{MLP_{best}}$, $\boldsymbol{Spread_{V3}}$ and $\boldsymbol{Dense_{BN}}$ trained with 40 000 training traces from *RD* using HW and intermediate value as leakage model.

regarding the countermeasures. First we briefly discuss the experimental results for $ASCAD_M$, and afterward, continue with the results for *RD*.

In Figure 4.9 we depict the guessing entropy of the architectures trained with 40 000 traces for both leakage models. For all networks we observe that $\boldsymbol{Spread_{PH}}$, $\boldsymbol{MLP_{best}}$, and $\boldsymbol{MLP_{RT}}$ do not recover the key, while $\boldsymbol{Spread_{V3}}$ and $\boldsymbol{Dense_{BN}}$ do. We believe these architectures are successful because they employ a batch normalization layer. Furthermore, we observe that $\boldsymbol{Dense_{BN}}$ requires fewer traces than $\boldsymbol{Spread_{V3}}$ for both leakage models. Note that we do not show the results when the architectures are trained with fewer traces since none of them are successful in retrieving the key. From these results, we believe that there is no need to employ a spread layer, or the improved $Spread_{L3}$, in a network when the traces are obtained from a masked implementation of AES. Despite $Spread_{L3}$ is able to recover the key, we believe its success is caused by the BN layer, which explains why $\boldsymbol{Dense_{BN}}$ is successful.

In Figure 4.10 we depict the results for *RD* trained with 40 000 traces for both leakage models. This figure shows that all architectures are not able to retrieve the key. Thus, similar to the masking countermeasure, we see no reason to employ a spread layer, or the improved $Spread_{L3}$ when the traces are obtained from an AES implementation with a random delay countermeasure.

4.5. Comparing Architectures

We have shown that the spread layer does not help to improve the attack efficiency for unprotected implementations. To evaluate if there is a need for a deep learning layer for unprotected implements, we compare the previously shown networks trained with 1 000 and 40 000 traces. The networks trained with 40 000 traces provide a baseline for how efficient an attack could be. Networks trained with 1 000 traces are used to make a comparison between the networks trained with 40 000 traces.

More specifically, we train the networks $\boldsymbol{MLP_{best}}$, $\boldsymbol{MLP_{RT}}$, and $\boldsymbol{Dense_{BN}}$ with 1 000 and 40 000 traces from the datasets $ASCAD_U$ and *DPAv4*. These networks are selected because they provided the best performance in the previously conducted experiments. The networks are trained for 80 epochs and trained with both HW and intermediate value as leakage model. Additionally, the learning rate is set to 10^{-4}, and the batch size to 100.

In Figure 4.11 we depict the results of the conducted experiments. In these figures, the lines with a star are trained with 1 000 traces and the lines with hexagons are trained with 40 000 traces. First we discuss the results for the $ASCAD_U$ dataset and then the results for the *DPAv4* dataset. The results with HW as leakage model show there is no significant difference in guessing entropy. As expected, all the networks trained with 40 000 traces perform the best, and networks trained with 1 000 traces are less efficient. However, notice that $\boldsymbol{Dense_{BN}}$ trained with 1 000 traces performs roughly the same as the networks trained with 40 000 traces. Similarly to the results from Hamming weights, $\boldsymbol{Dense_{BN}}$ trained with 1 000 traces performs near similar to the networks trained with 40 000 traces. Thus, the results when using intermediate value as leakage model show that the networks trained with 40 000 traces perform better.

The results from the attacks on *DPAv4* are similar to the results from the attacks on $ASCAD_U$. For networks trained on the Hamming weight, we observe a minimal difference in attack efficiency and see that the networks trained with 40 000 traces perform the best. Networks trained on the intermediate value show a bigger difference. The networks trained with the most traces retrieve the key in a single trace, while the networks trained with 1 000 traces require around 15 traces.

For all the conducted experiments in this section we observed there is a minor improvement in the attack efficiency when increasing the training size. Therefore, developing a deep learning layer specifically for side-channel attacks on unprotected implementations would only improve the attack efficiency by a small margin. Because of this reason we believe there is no need to develop a deep learning layer for side-channel attacks on unprotected implementations.

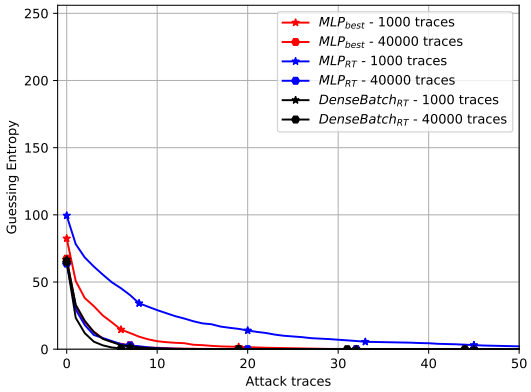
4.6. Conclusions

In this chapter, we have first described the spread layer, and subsequently provided an analysis of both SCA performance and in-and-output activations. For the SCAs with $\boldsymbol{Spread_{PH}}$, we have shown that a comparable architecture can achieve similar results and thus the spread layer is not necessary. Furthermore, we noticed that the spread layers' hyperparameters, the spread factor, does not have a significant influence on the guessing entropy.

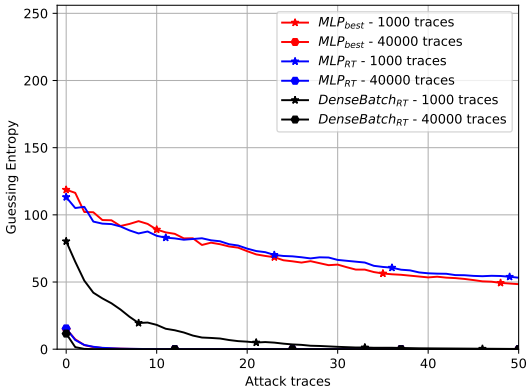
Therefore we have analyzed the spread layers' in-and-output activations, in which we showed that the spread layer did not work as intended. The spread layer intends to spatially spread its input, which does not occur in practice. We have shown that the spread layer's output consists mostly of non-active neurons. The cause of this problem is two-fold: 1) the minimum and maximum used to remap its input are learned incorrectly and 2) the spread layer expects its input to be uniformly distributed while in reality it is normally distributed.

The first issue is caused by how the spread layer learns the minimum and maximum. We have fixed this issue with the layer $Spread_{L2}$ and is utilized in the architecture $\boldsymbol{Spread_{V2}}$. The second issue is caused by a difference of the expected input distribution. To fix both of the issues we have introduced the layer $Spread_{L3}$ and is utilized in the architecture $\boldsymbol{Spread_{V3}}$. To compare both architectures we introduced a new architecture $\boldsymbol{Dense_{BN}}$.

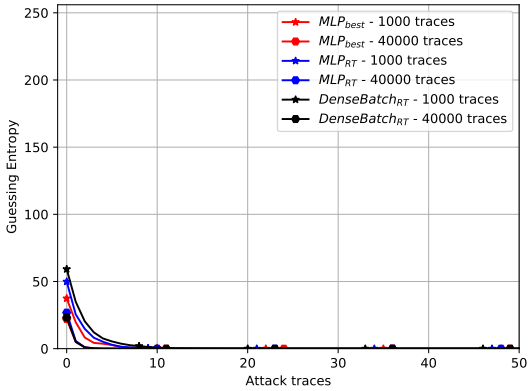
For our comparisons, we have experimented mostly with settings for which around 1 000 traces were used in the profiling phase, given that the spread layers' most remarkable result was that a limited amount of traces are required during the profiling phase. In our experiments, we have shown that the spread layer does not improve SCAs for both unprotected and protected implementations by comparing $\boldsymbol{Spread_{PH}}$ with comparable architectures. Even architectures that used our proposed fixes for the spread layer, i.e. $\boldsymbol{Spread_{V2}}$ and $\boldsymbol{Spread_{V3}}$, did not improve the attack in comparison with $\boldsymbol{Dense_{BN}}$. Therefore we conclude that the spread layer does not improve the performance for SCAs in both protected and unprotected settings.



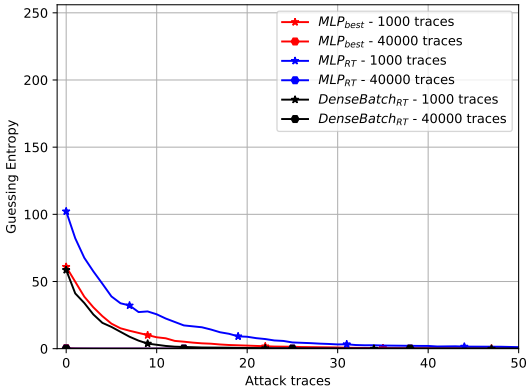
(a) Guessing entropy of $ASCAD_U$ using HW model



(b) Guessing entropy of $ASCAD_U$ using intermediate value



(c) Guessing entropy of *DPAv4* using HW model



(d) Guessing entropy of *DPAv4* using intermediate value

Figure 4.11: Guessing entropy of networks trained with 1 000 and 40 000 traces from $ASCAD_U$ and *DPAv4* using HW and intermediate value as leakage model.

Additionally, we selected the best performing networks to investigate the difference in attack efficiency when using a small and big training size. We show that the difference between networks trained with many traces performs almost similar to networks trained with fewer traces. The gap is so small such that there is no incentive to develop a new layer. For all datasets we do no observe any need for specific layers, thus the already known layers are sufficient for SCA.

5

Evaluation of the Kernel Size and Depth of CNNs

Recent works have shown that convolutional neural networks perform well for side-channel attacks even for protected implementations where countermeasures are present. For example, in [25], the authors use a CNN architecture specifically designed for a different domain than SCA and show it achieves state-of-the-art performance on various datasets. However, CNNs have many hyperparameters that are required to be properly configured for efficient attacks, but there is limited knowledge about the influence of the hyperparameters on the attack efficiency for SCAs.

Typically, the configuration of hyperparameters for SCAs using CNNs is influenced by recommendations from the image recognition domain. However, side-channel measurements obtained from protected implementations have different characteristics than images. More specifically, images do not have a countermeasure and do not actively attempt to harden the classification task. Therefore it is important to evaluate if the recommendations from the image recognition domain hold as well for the side-channel analysis domain. In this chapter, we will evaluate the influence on the attack efficiency of two hyperparameters of CNNs, namely, the kernel size and network's depth.

This chapter is outlined as followed. First, we provide an in-depth analysis of why we experiment with the kernel size and depth of a network. Before we show and discuss the results, we describe the CNN architectures used for the experiments. Then, we discuss the experimental settings such as the configurations of the hyperparameters. After this, we depict and discuss the experimental results. Note that we make a difference between the hiding and masking countermeasure and discuss the results individually. Finally, we provide a conclusion of the observed results.

5.1. Motivation

The universal approximation theorem states that a feed-forward network with a single hidden layer and a finite amount of neurons can approximate any function under certain conditions [20]. However, the number of neurons required to achieve this is exponential and thus is unfeasible to employ in practice. Additionally, it is assumed that increasing a CNN's depth, increases the network's ability to approximate the target function. This idea is further supported by what has been shown in research, where deep CNNs achieve state-of-the-art performance, such as VGG and Inception [24].

Furthermore, what is observed for the state-of-the-art CNNs is that the kernel size is kept small, and is no larger than 11. It is argued that a small kernel can recognize fine-grained detail, while a large kernel size recognizes coarse-grained detail. It is thought that a small kernel can recognize lines and edges that occur more often. Additionally, it is thought that the deeper layers in a CNN, use the learned fine-grained details to recognize bigger objects [24]. This is important for the domain of image recognition since typical images exist of many details and objects.

However, images are not similar to measurements for side-channel attacks, especially if they are obtained from protected implementations. The key difference is that measurements from protected implementations have one or more countermeasures that harden the classification task at hand, which does not occur in images. For both countermeasure, masking and random delay, we intuitively explain what we expect to be the

influence of the kernel size and depth of a CNN[1].

For the random delay countermeasure, we argue that a large kernel size improves the attack efficiency. Intuitively, recognizing small details for the random delay countermeasure might not be as important as eliminating the random delay countermeasure. Therefore, employing a large kernel size could potentially improve attack efficiency. To explain this imagine a set of traces X, where we represent a single trace as $\vec{x} = \{x_0, x_1, \ldots, x_n\}$. For simplicity, we assume that there is a single feature x_a that leaks information. For the random delay countermeasure, we assume the index of this feature is in some bounds $i \leq a \leq j$. By using a large kernel we aim to have an anchor point (that is before index i) from which we can detect even the largest value of a random delay. Thus if the kernel size is set to $k = j - i$, we could possibly eliminate the effect of the countermeasure. Since the values of i and j are unknown in normal settings, the best kernel size has to be found by experimental results. Additionally, deep CNNs with a small kernel have already been shown to be efficient against a random delay countermeasure.

The masking countermeasure is different, but we believe a large kernel or deep network could perform an efficient attack. To explain this, we assume there are two leakage points, one for the mask and the processing of the mask, additionally, they are always located in the same position. If a network can learn a filter that bridges this distance and can pinpoint the mask and processing of the mask, we can attack a masked implementation. Like random delay, deeper networks with a small kernel size first recognize small patterns and then the bigger patters. Thus, the first layers could recognize the mask and the processing of it, and then combine the two findings to perform the classification task.

Because of the provided reasons we analyze the influence of the kernel size and the depth of CNNs. For both discussed countermeasures, hiding and masking, we perform experiments and evaluate the results. Additionally, we compare the performance trade-off between the depth and kernel size of a CNN.

5.2. Experimental Setup

To analyze the influence of the kernel size and depth of the network, we should first choose a CNN architecture to conduct the experiments with. In [25], the authors use a VGG-like network that achieves state-of-the-art performance on the *RD* dataset. In this chapter we will use a modification of this network, with the main difference between the architecture presented in [25], is that we vary the number of convolutional layers in the network. We employ less convolutional layers because of computational constraints: large kernel sizes in combination with many convolutional layers significantly increases the training time and memory usage, making it unfeasible to analyze. Next to this, we do not implement the addition of Gaussian noise to reduce the probability of overfitting since we try to isolate the influence of kernel size and the number of layers in a convolutional block.

To be more precise about the architecture we implement, we first provide definitions we use to describe the implemented architecture:

Definition 5.2.1. A *convolutional block* (CB) of l layers consists of the sequence: l convolutional layers with ReLU as activation function, and subsequently a max-pooling and batch normalization layer. The convolutional layers use the same hyperparameters.

Definition 5.2.2. A *feature selection block* consists of three convolutional blocks of l layers, followed by a flatten layer. The convolutional blocks use the same hyperparameters except for the number of kernels, which is doubled in each convolutional block.

Definition 5.2.3. A *classification block* consists of the sequential layers: dropout, MLP with ReLU as activation, dropout, MLP with softmax as activation. The MLP layers have the same amount of neurons, except for the output neurons which depend on the leakage model.

The architecture for the experiments consists of a feature selection block with two parameters l and k, and a classification block. We denote the architecture as $\boldsymbol{VGG}_{l,k}$ where l is the number of convolutional layers in a convolutional block, and k the kernel size of the convolutional layers. The hyperparameter configurations for the feature selection and classification block are shown in Table 5.1a and Table 5.1b respectively. In these tables, we only denote the number of kernels in the first convolutional block, and each subsequent block doubles the number of kernels. Thus, in the first convolutional block, there are 32 kernels, the second 64, and the last 128. Figure 5.1 depicts the architecture of $\boldsymbol{VGG}_{2,7}$ in which there are two convolutional layers in per block and the kernel size is fixed to 4.

[1] Note that, we consider a small kernel size as a value not higher than 11, larger values are considered as a large kernel (since this is typically not observed in state-of-the-art CNN architectures).

#Kernels	Pool	Padding	Stride	Dilation
32	2	same	1	1

(a) Feature selection block configuration

#Neurons	Dropout	Leakage model
256	$p = 0.5$	ID/HW

(b) Classification block configuration

Table 5.1: Architecture configuration

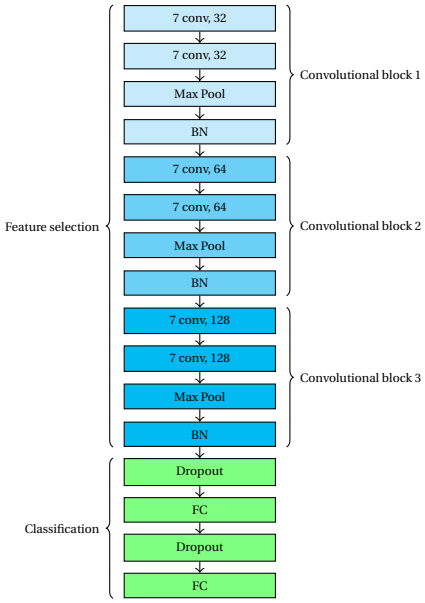


Figure 5.1: A sample architecture with kernel size 7, and 2 convolutional layers in a convolutional block. The first convolutional block has 32 filters, the second 64, and the last 128.

There is however a problem with the networks in certain conditions: when the input space is large, the output of the feature selection block consists of many neurons. As a result, the subsequent layer has a factor of the number of neurons more parameters, which decreases the probability of the network converging. To counter this we either apply L2-regularization or reduce the number of neurons by increasing the max pool size. We will mention for which experiments L2-regularization is applied. For the increased pooling factor we introduce a new architecture and denote it as $\boldsymbol{VGGMax}_{l,k}$, where l and k are the number of convolutional layers in a block and the kernel size respectively. The configurations are similar to those denoted in Table 5.1, but the pooling size is increased from two to four. The other hyperparameters used for training are shown in Table 5.2.

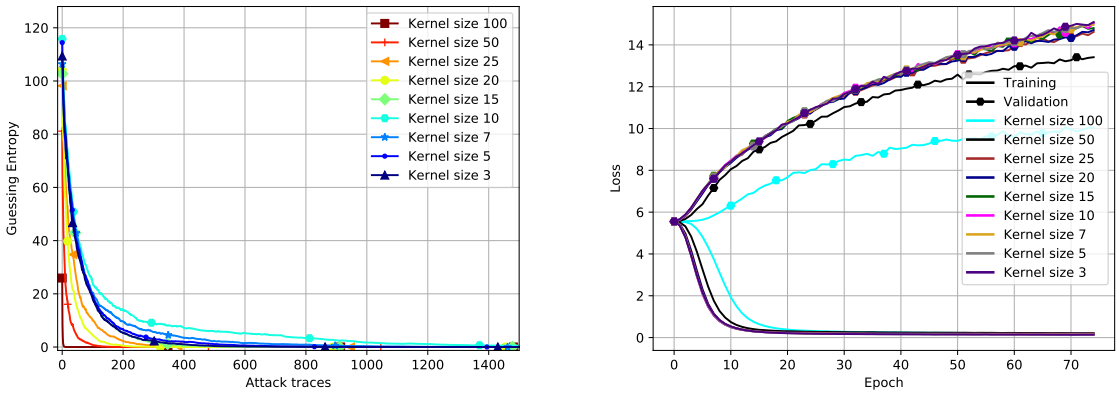
Since we will be experimenting with neural networks with many layers the initialization is important for convergence (a network is not able to learn anything if it does not converge). As discussed in chapter 3, for networks with many convolutional layers using ReLU, Kaiming initialization produces better results than the standard Xavier initialization. We have experimented with both types of initialization and noticed that networks initialized using Xavier's method showed convergence problems, while models initialized with Kaiming initialization would always converge. Therefore, for the experiments run in this chapter, we will use Kaiming initialization.

In order to evaluate the attack efficiency of a configuration, we learn five folds using the mentioned configurations. We have chosen to train five folds because of the consideration between computational constraints and the number of models that will provide a sufficient view of the performance. For each of the trained models, we calculate individually the *PGE* and average the results. Then, with the average we can, for example ,determine if the attack is successful with the configurations or after the amount of traces required to recover the key.

For some experiments we have performed in this chapter, there is a minimal difference in *PGE*. In these cases either the dataset is to easy or the model just works really well for the dataset. In order to find the difference in performance between the configurations, we will look at the test accuracy and perform a trick

Batch size	LR	Epochs	Training size	Channel size	Initialization
100	1e−4	75	40 000	32	Kaiming

Table 5.2: Hyperparameter configurations for the CNN experiments



(a) *PGE* (b) Validation loss during training

Figure 5.2: *PGE* and validation loss of $\boldsymbol{VGG}_{2,k}$ on the dataset *RD*

with the test traces. By adding noise to the test traces we make it harder for the models to perform the attack. By doing so, we hope to notice a difference between different configurations. Next to this, in a more realistic profiling attack (where the profiling device and target device are not the same devices) there would be some difference between the profiling and attack traces, and adding noise to the attack traces would make it a more realistic setting. The noise that is added to the attack traces is drawn from a Gaussian distribution $N(0,1)$ and multiplied by a factor f. Thus, we create various sets of attack traces, with each a different level of noise, calculated by $N(0,1) \cdot f$.

5.3. The Hiding Countermeasure

In this section, we will experiment with datasets in which the random delay countermeasure is present. The datasets we will consider are *RD*, $ASCAD_D^{50}$, and $ASCAD_D^{100}$. With the experiments we perform in this section we aim to answer the research questions **RQ 2.**, **RQ 3.**, and **RQ 4.**.

5.3.1. Experimental Results Kernel Size

To answer **RQ 2.** we experiment with the architecture and hyperparameters discussed in the previous section. For our experiments, we will use the architecture $\boldsymbol{VGG}_{2,k}$ with two convolutional layers in a block and $k \in \{3,5,7,10,15,20,25,50,100\}$. We have chosen to use two convolutional layers in a convolutional block because of computational constraints. If we use two layers we can still select large kernel sizes, for example 100, such that the experiments take a reasonable time to finish.

Dataset *RD*

For the experiments performed in this section, we use measurements from the *RD* dataset. For the first experiment we discuss, we have trained $\boldsymbol{VGG}_{2,k}$ with the configurations discussed in the previous section (see Table 5.1 and Table 5.2). The results of this experiment are depicted in Figure 5.2. This figure shows a plot of the *CGE* and a plot of the validation loss during training for all kernel sizes. From Figure 5.2a the influence of the kernel size is visible; an increase in the kernel size results in a lower *CGE*. For example, $\boldsymbol{VGG}_{2,3}$ requires around 109 times more traces to recover the key than $\boldsymbol{VGG}_{2,100}$. If we analyze the validation loss, depicted in Figure 5.2b, we observe that all the experiments overfit massively, however, the architectures $\boldsymbol{VGG}_{2,50}$ and $\boldsymbol{VGG}_{2,100}$ overfit the least. This is remarkable since naturally we would expect that the models would overfit even more because of the additional parameters. The results depicted in Figure 5.2 are the first indication that large kernel sizes are highly effective against the random delay countermeasure.

Kernel size	3	5	7	10	15	20	25	50	100
Accuracy	1.36	3.33	15.23	32.23	44.63	54.93	48.05	61.85	44.11

Table 5.4: Accuracy of $\boldsymbol{VGG}_{2,k}$ trained with $\lambda = 0.05$ on *RD*

Noise (α)	**Kernel size**								
	3	5	7	10	15	20	25	50	100
0.0	73	28	18	12	12	10	9	8	7
0.25	123	47	38	18	14	11	12	8	7
0.5	546	311	172	112	90	54	65	21	22
0.75	-	1 737	2 048	1 264	1 105	653	897	130	139
1.0	-	-	-	-	-	-	-	-	977

Table 5.6: Results for $\boldsymbol{VGGMax}_{2,k}$ with added noise on *RD*

To counter the problem of overfitting we conduct two more experiments to reduce the overfitting: 1) apply L2-regularization and 2) train models with $\boldsymbol{VGGMax}_{2,k}$. For L2 regularization we use the following L2-penalties $\lambda \in \{0, 0.05, 0.005\}$, where an L2-penalty of zero is the original experiment. In Figure 5.3, we show plots of the *PGE* for the experiments where L2 regularization is applied, and in Figure 5.3 we list the *CGE* for all kernel sizes. For the experiments where $\lambda = 0.0$ and $\lambda = 0.005$, this table shows that an increase in the kernel size results in a lower *CGE*. However, for the experiment where an L2-penalty of 0.05 was applied the difference in *CGE* is minimal, for example, $\boldsymbol{VGG}_{2,10}$ and $\boldsymbol{VGG}_{2,50}$ have the same *CGE*, namely 5. By analyzing the accuracy of the attack set of $\boldsymbol{VGG}_{2,k}$ the difference between the kernel sizes are more evident, we list the accuracies of this experiment in Table 5.4. The highest accuracy is achieved by $\boldsymbol{VGG}_{2,50}$, which has around 60% correct. The lowest accuracy is achieved by $\boldsymbol{VGG}_{2,3}$, which has around 1% correct. Surprisingly, $\boldsymbol{VGG}_{2,100}$ has a lower accuracy than $\boldsymbol{VGG}_{2,50}$. We believe a kernel size of 100 is too high (with the used hyperparameters), which causes the network to overfit.

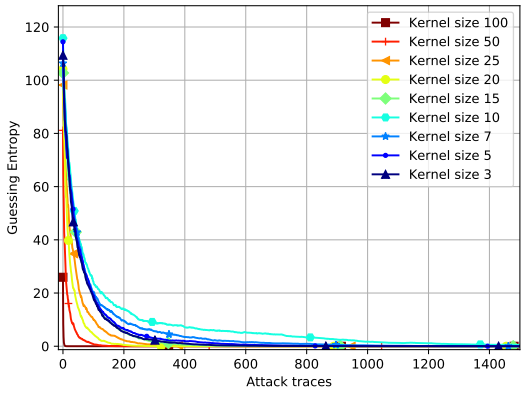
Since accuracy is not the best metric for SCA and we still want to highlight the difference in the performance of the various kernel sizes, therefore we add noise to the attack traces and evaluate the classifiers again using the noisy traces. To do so we use the previously discussed method and create four new attack sets with distinct noise levels. More specifically, the noise levels are $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$. We depict the *PGE* of all the noise levels in Figure 5.4. Additionally, in Table 5.5 we list the *CGE* of each of the noise levels. The figure clearly shows that increasing levels of noise has less effect on the *PGE* if a network architecture employs a larger kernel size. For extremely noisy data where $\alpha = 1.0$, we observe that only $\boldsymbol{VGG}_{2,100}$ can retrieve the key consistently. Furthermore, for all noise levels, we notice that the *CGE* is declining if the kernel size is increased.

Next to applying L2-regularization to prevent overfitting, we perform experiments with $\boldsymbol{VGGMax}_{2,k}$ using the same settings as previously discussed. For these experiments, we only show the *CGE*, which is listed in Table 5.6. Again, we observe that an increase in the kernel size results in a decrease in *CGE*. However, the results in the upper range of the tested kernel sizes are similar, therefore, we add noise to the attack traces with $\alpha \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The experiments with the noisy traces make the gap of *CGE* between two adjacent kernel sizes bigger. Here we see that a model learned with a large kernel size is still able to recover the key, and is thus more robust to noisy data.

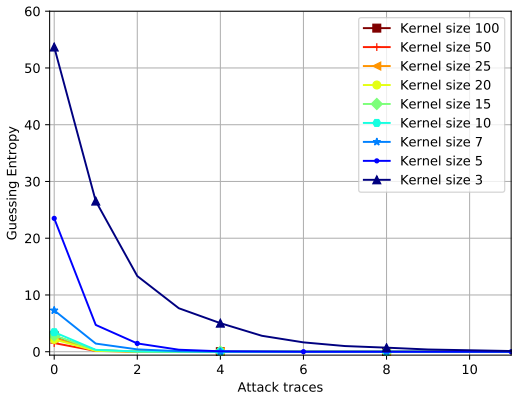
Datasets $\boldsymbol{ASCAD}_D^{50}$ and $\boldsymbol{ASCAD}_D^{100}$

The next datasets we consider are the two variants of the ASCAD database for which the authors have added an artificial random delay. More specifically, we will consider the datasets $ASCAD_D^{50}$ and $ASCAD_D^{100}$. For the experiments, we will use the same configurations as for the *RD* dataset, however, here we will use HW as the leakage model since this results in more efficient attacks. For the performed experiments we observe minimal overfitting and thus we do not apply L2-regularization or use $\boldsymbol{VGGMax}_{2,k}$.

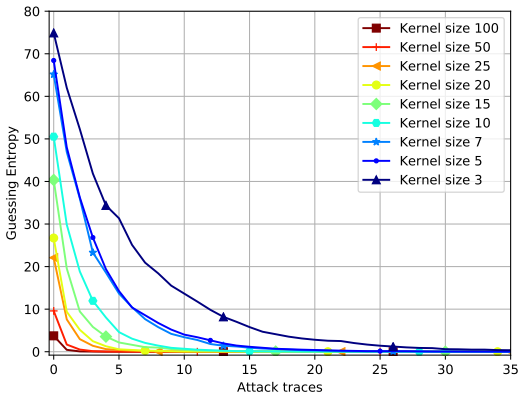
In Table 5.7 and Table 5.8 the results are presented of datasets $ASCAD_D^{50}$ and $ASCAD_D^{100}$ respectively. In these tables we present the *CGE* of the experiments, if a network was not able to recover the key we denote this with a dash. First, we discuss the results of the conducted experiments of the dataset $ASCAD_D^{50}$ followed by the results of the dataset $ASCAD_D^{100}$.



(a) *PGE* when $\lambda = 0.0$



(b) *PGE* when $\lambda = 0.05$

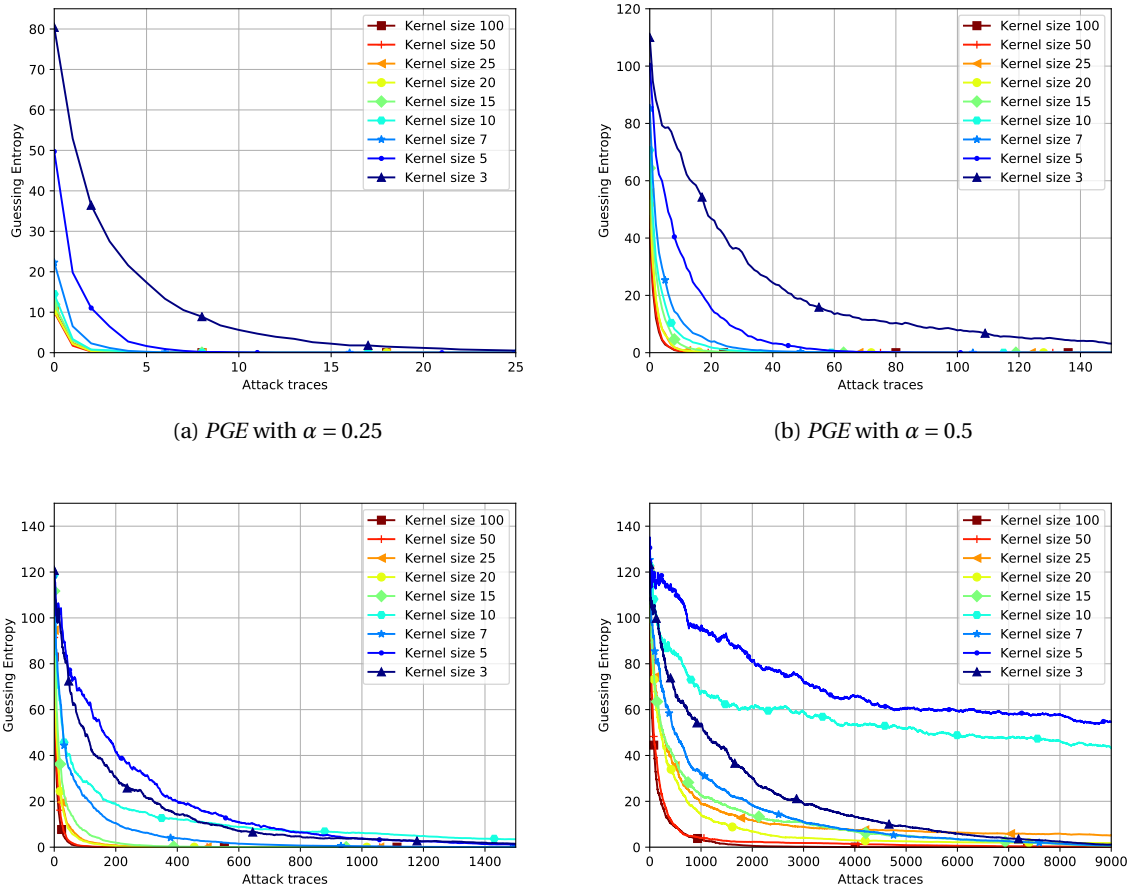


(c) *PGE* when $\lambda = 0.005$

Figure 5.3: *PGE* for two convolutional layers in a block of $\boldsymbol{VGG}_{2,k}$ with dataset *RD*

L2-penalty (λ)	**Kernel size**								
	3	5	7	10	15	20	25	50	100
0	1 636	1 495	1 797	2 757	999	447	973	224	15
0.05	24	8	6	5	5	3	5	5	4
0.005	89	41	43	27	32	22	21	8	6

Table 5.3: *CGE* for $\boldsymbol{VGG}_{2,k}$ with different L2-penalties on *RD*



(a) *PGE* with $\alpha = 0.25$

(b) *PGE* with $\alpha = 0.5$

(c) *PGE* with $\alpha = 0.75$

(d) *PGE* with $\alpha = 1.0$

Figure 5.4: *PGE* for adding various noise factors $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$, and $\lambda = 0.05$ ($\boldsymbol{VGG}_{2,k}$ on *RD*)

Noise (α)	**Kernel size**								
	3	5	7	10	15	20	25	50	100
0.25	45	17	13	10	8	10	7	9	6
0.5	688	159	133	110	53	47	45	26	37
0.75	4 173	3 921	2 152	8 883	779	702	740	310	329
1.0	-	-	-	-	-	-	-	-	5 609

Table 5.5: Results for $\boldsymbol{VGG}_{2,k}$ trained with $\lambda = 0.05$ and added noise to attack set on *RD*

Noise (α)	**Kernel size**								
	3	5	7	10	15	20	25	50	100
0.0	49	28	24	32	25	37	21	27	16
0.1	123	53	51	44	45	49	47	53	52
0.2	-	2 568	433	187	197	261	257	351	467
0.3	-	-	-	1 136	865	1 205	1 167	2 069	-
0.4	-	-	-	-	2 666	4 756	3 544	9 969	-

Table 5.7: Results for $\boldsymbol{VGG}_{2,k}$ using dataset $ASCAD_D^{50}$ with added noise to attack traces.

Noise (α)	**Kernel size**								
	3	5	7	10	15	20	25	50	100
0.0	31	38	35	31	29	37	27	22	15
0.1	80	137	100	66	67	91	72	73	95
0.2	-	-	-	408	843	1 201	3 274	903	2 466
0.3	-	-	-	9 960	-	-	-	-	-
0.4	-	-	-	-	-	-	-	-	-

Table 5.8: Results for $\boldsymbol{VGG}_{2,k}$ using dataset $ASCAD_D^{100}$ with added noise to attack traces.

For the conducted experiments with $ASCAD_D^{50}$, we observe that smallest kernel size performs the worst and the largest performs the worst. However, the difference in *CGE* between the kernel sizes between the largest and smallest is minimal. Therefore, we add noise to the attack traces where the noise levels are $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$. The results in Table 5.7 show that as the noise increases, the networks with a larger kernel size are able to recover the key, while the smaller ones do not. For example, when $\alpha = 0.2$ the smallest kernel that is able to recover the key is 5, while in the setting where $\alpha = 0.3$ this is 10. In terms of efficiency, we observe that for high noise settings $\alpha = 0.3$ and $\alpha = 0.4$, the most efficient networks of the networks which can recover the key, are the ones with smallest kernel.

The results with dataset $ASCAD_D^{100}$ are somewhat similar as shown in Table 5.8. The best performing network is the one with the biggest kernel, and the worst performing is the one with the second to smallest kernel. Like the experiments for $ASCAD_D^{50}$, we perform additional experiments with noise added to the attack traces, where $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$ because the difference in *CGE* is minimal. Unlike the result for $ASCAD_D^{50}$, in the highest noise setting ($\alpha = 0.4$), no network is able to recover the key. We believe this difference is caused because $ASCAD_D^{100}$ has a bigger artificial delay, making this dataset somewhat harder. With a lower noise level $\alpha = 0.3$, the only network to recover a key is the network with a kernel size of 10. With lower levels of noise we observe that the most efficient networks are the ones with a kernel size of 10.

For the three discussed datasets we have experimentally shown the influence of kernel size by evaluating the performance of the learned classifiers. In general, for the discussed classifiers, we have shown that *CGE* decreases when the kernel size is increased. In some cases when the difference in *CGE* was minimal, we have used noisy attack traces to enlarge the difference. Here, the results showed that a larger kernel can achieve more efficient attacks. However, for the ASCAD database we noticed that at some point the performance stagnates and there is no benefit from increasing the kernel size even further. Despite these findings, we still observe that, in general, a larger kernel size achieves more efficient attacks.

5.3.2. Experimental Results Stacked Convolutional Layers

In this section, we will conduct experiments that allow us to evaluate the influence of the number of convolutional layers in a convolutional block. The evaluation provides us with more insights such that we can answer **RQ 3.**. From now on, if we refer to the number of layers, we refer to the number of convolutional layers in a convolutional block. For the experiments we use $\boldsymbol{VGG}_{l,15}$ where $l \in \{1, 2, 3, 4, 5\}$. The kernel size is fixed to 15 because in the previous section we showed that larger kernel sizes result in more efficient attacks, however, a kernel size much larger than 15 makes the experiments infeasible because of computational constraints. The hyperparameters used for training the models are equal to the ones used in the discussion about the kernel size and are listed in Table 5.1 and Table 5.2. The datasets we will consider are the three datasets where the

hiding countermeasure is present, namely *RD*, $ASCAD_D^{50}$, and $ASCAD_D^{100}$.

Dataset *RD*

The experimental results of SCAs using the architecture $\boldsymbol{VGG}_{15,l}$ with measurements from the *RD* dataset are depicted in Figure 5.5a. This figure clearly shows that an increase in the number of layers results in a decrease of the *CGE*. The difference between using one layer and five layers is quite significant; when $l = 5$ around 16 traces are required to recover the key, while when $l = 1$ we need around 802 traces.

In Figure 5.5b and Figure 5.5c we depict the results when using $\lambda = 0.05$ and $\lambda = 0.005$ respectively. Both figures show that an increase in the number of layers has a positive effect on the guessing entropy. However, the results in guessing entropy are almost similar and require at most 20 traces to recover the key. For $\lambda = 0.05$ we observe that we require at least 3 traces and at most 8 traces to recover the key. For $\lambda = 0.005$, the difference between the minimum and maximum guessing entropy is slightly larger, where we need at least 3 traces and at most 42 traces to recover the key. Here, the results indicate that an increase in the depth of the network decreases the amount of traces required to recover the key. Moreover, we list the *CGE* in Table 5.9 for all λ.

To highlight the difference between the resulting guessing entropy of the architectures, we add noise to the attack traces. We conduct this experiment on the models trained with $\lambda = 0.05$ since for these models the difference in guessing entropy is the least. The noise levels we perform experiments on are $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$. In Figure 5.6 we depict guessing entropy plots for $\alpha = 0.75$ and $\alpha = 1.0$, and in Table 5.10 we list the *CGE* for all noise levels. We only depict the results for $\alpha = 0.75$ and $\alpha = 1.0$ in Figure 5.6, since these results highlight the difference between the number of layers in an architecture.

In the most extreme case, when $\alpha = 1.0$, we observe that the only model that can consistently recover the key is when a single layer is used. Other values of α show similar behavior; increasing the depth of the network is not beneficial for the guessing entropy. This is in contrast of the experiments when $\lambda = 0.0$ and $\lambda = 0.005$. Further analysis is required to explain these discrepancies, which will be performed in the subsection 5.3.3.

Additionally, we conduct experiments with the architecture ***VGGMax***, with a fixed kernel size of 15 and 4 as max-pooling value. The *CGE* for all values of l shows no significant difference since the *CGE* is relatively low. At most 13 traces and at least 5 traces are required to recover the key. Because the values of *CGE* are so close together we perform experiments with the noisy attack traces. Like the previous experiments, we experiment with various noise factors where $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$. In Table 5.11 we list the *CGE* for $\boldsymbol{VGG}_{l,15}$ with and without noise factors. In Figure 5.7 we depict the *PGE* of the noise factors $\alpha = 0.75$ and $\alpha = 1.0$ since these noise levels highlight the biggest difference between the number of layers in the network. From the results in the figures and table, we observe that increasing the number of layers decreases the *CGE* until $l = 4$. Increasing the number of layers even further shows a increase in *CGE*. We believe this occurs because of vanishing gradients.

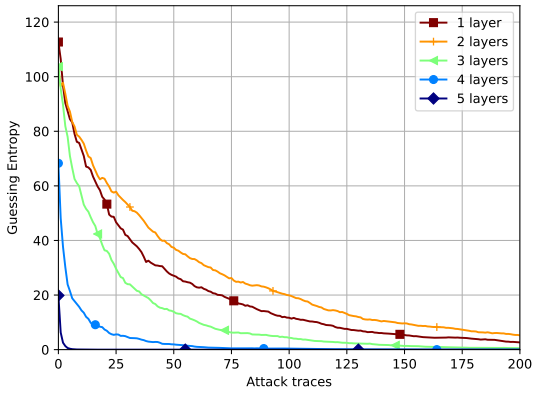
Datasets $ASCAD_D^{50}$ and $ASCAD_D^{100}$

Here we depict and discuss the results for an attack on the ASCAD database for which an artificial delay has been added to the traces. For these datasets, we will only experiment with $\boldsymbol{VGG}_{l,15}$, and we will first consider the dataset $ASCAD_D^{50}$. In Table 5.12 we list the resulting *CGE* for this experiment. Since the *CGE* is all close to each other, we attack the noisy datasets with noise factor $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$. The plots of the *CGE* are depicted in Figure 5.8. The results depicted in these figures show that only the architecture where $l = 1$ is not able to retrieve the key if $\alpha > 0.1$. Thus, for this dataset, deeper network achieves better results.

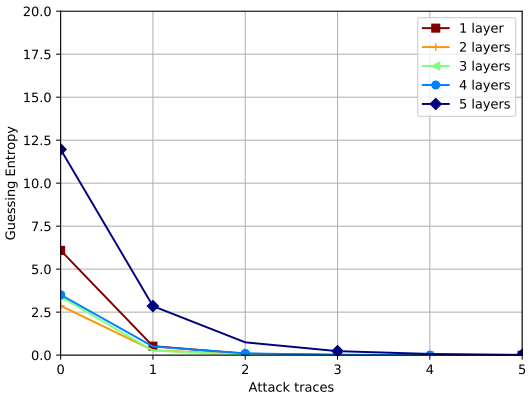
For $ASCAD_D^{100}$ we notice similar behavior when using the original traces (thus when $\alpha = 0$), the *CGE* is similar for all values of l. Performing the attack with noisy trace, we observe a bigger difference in *CGE*. In Figure 5.9 we depict the plots of the *PGE* of the different noise factors α, and in Table 5.13 we list the *CGE*. We observe the first difference between $ASCAD_D^{50}$ and $ASCAD_D^{100}$ when $\alpha = 0.2$, here we see that $\boldsymbol{VGG}_{2,15}$ requires around 4 times more traces to recover the key than deeper architectures. When adding more noise, $\alpha = 0.3$, we observe that $\boldsymbol{VGG}_{2,15}$ is not able to retrieve the key, while the deeper networks are able to recover the key. The largest difference we observe is for the noise factor $\alpha = 0.4$. For this noise factor, we observe that only $\boldsymbol{VGG}_{4,15}$ and $\boldsymbol{VGG}_{5,15}$ can recover the key, and $\boldsymbol{VGG}_{5,15}$ performs the best of these.

From these observations, we conclude that deeper networks are more robust than shallow ones. The difference between the reported results in this section of $ASCAD_D^{50}$ and $ASCAD_D^{100}$ can be explained by the bigger length of the artificial delay of the datasets. If the random delay is larger, increasing the depth of the network increases the attack efficiency when using noisy traces.

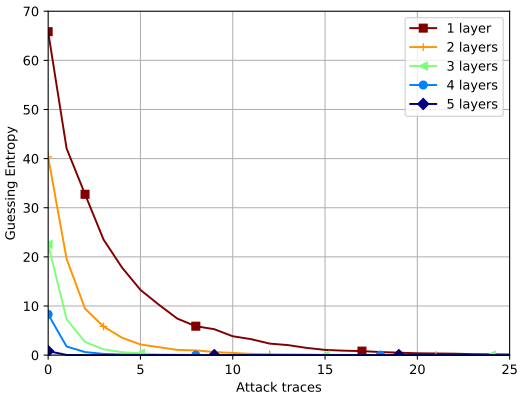
In general, from the three datasets we have considered in this section, we have experimentally shown that deeper networks provide better results in terms of guessing entropy. We have shown this by analyzing



(a) *PGE* when $\lambda = 0.0$



(b) *PGE* when $\lambda = 0.05$

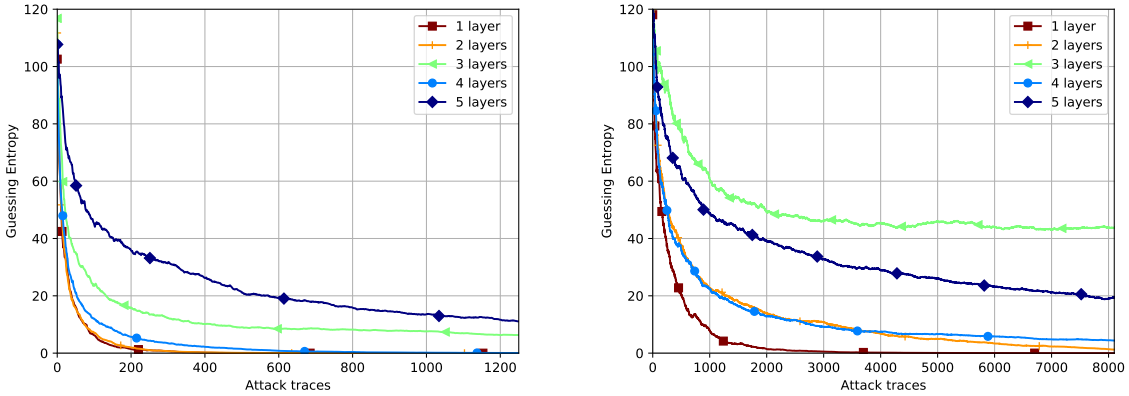


(c) *PGE* when $\lambda = 0.005$

Figure 5.5: *PGE* of $\boldsymbol{VGG}_{l,15}$ where $l \in \{1,2,3,4,5\}$ and trained with L2-penalty $\lambda \in \{0.0, 0.05, 0.005\}$ on the dataset *RD*

L2-penalty (λ)	**Conv layers in a block**				
	1	2	3	4	5
0.0	802	999	568	284	16
0.05	5	5	3	5	8
0.005	42	32	19	11	3

Table 5.9: Results for $\boldsymbol{VGG}_{l,15}$ trained with various L2-penalties on *RD*.

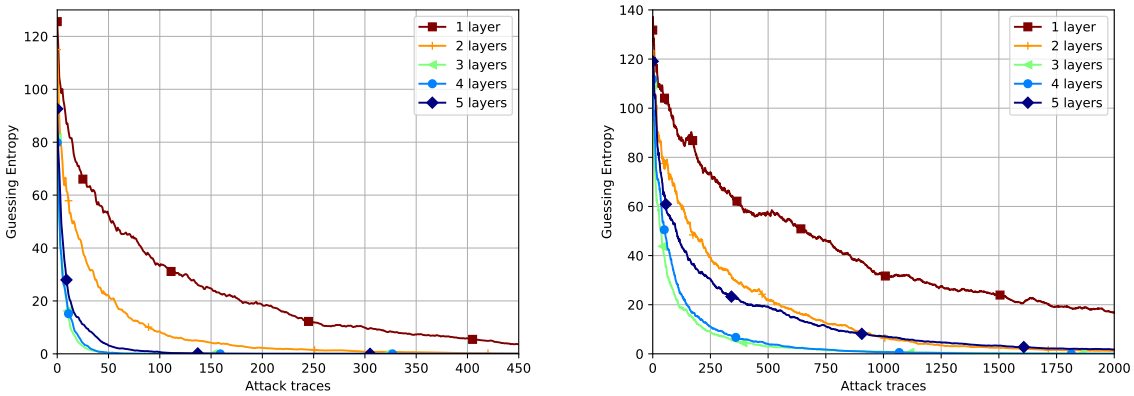


(a) *PGE* with $\alpha = 0.75$ (b) *PGE* with $\alpha = 1.0$

Figure 5.6: *PGE* of models trained with $\lambda = 0.05$ while attacking noisy traces with $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$ for various number of convolutional layers in a block on *RD*

Noise (α)	**Conv layers in a block**				
	1	2	3	4	5
0.25	15	9	7	10	22
0.5	89	76	91	69	539
0.75	904	1 082	-	1 839	-
1.0	8 085	-	-	-	-

Table 5.10: Results for $\boldsymbol{VGG}_{l,15}$ trained with L2-penal 0.05 and added noise to attack set on *RD*

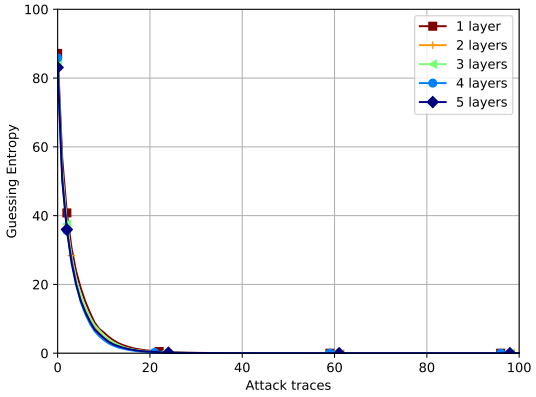


(a) *PGE* with $\alpha = 0.75$ (b) *PGE* with $\alpha = 1.0$

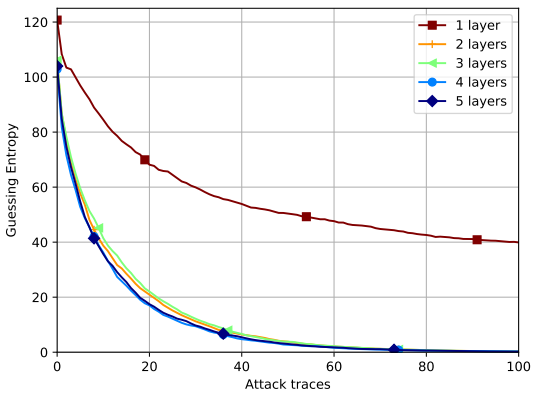
Figure 5.7: *PGE* of $\boldsymbol{VGGMax}_{l,15}$, with $l \in \{1,2,3,4,5\}$, while using noisy attack traces $\alpha \in \{0.75, 1.0\}$ on the dataset *RD*.

Noise (α)	**Conv layers in a block**				
	1	2	3	4	5
0.0	13	12	5	12	8
0.25	24	19	7	11	7
0.5	187	82	25	29	28
0.75	1 593	778	155	148	289
1.0	-	6 006	5 088	4 230	7 785

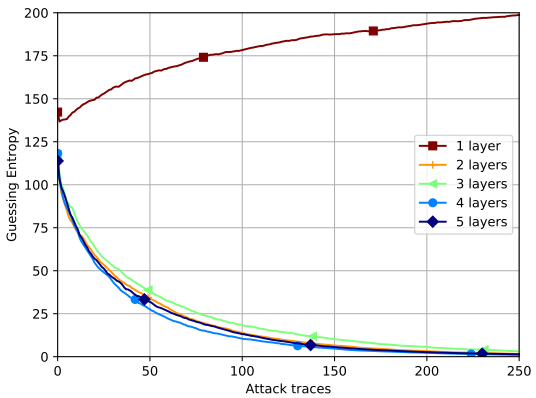
Table 5.11: Results for $\boldsymbol{VGGMax}_{l,15}$ and added noise to attack set on *RD*.



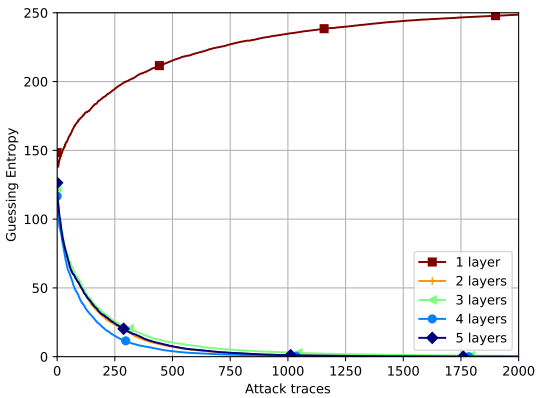
(a) *PGE* when $\alpha = 0.1$



(b) *PGE* when $\alpha = 0.2$



(c) *PGE* when $\alpha = 0.3$

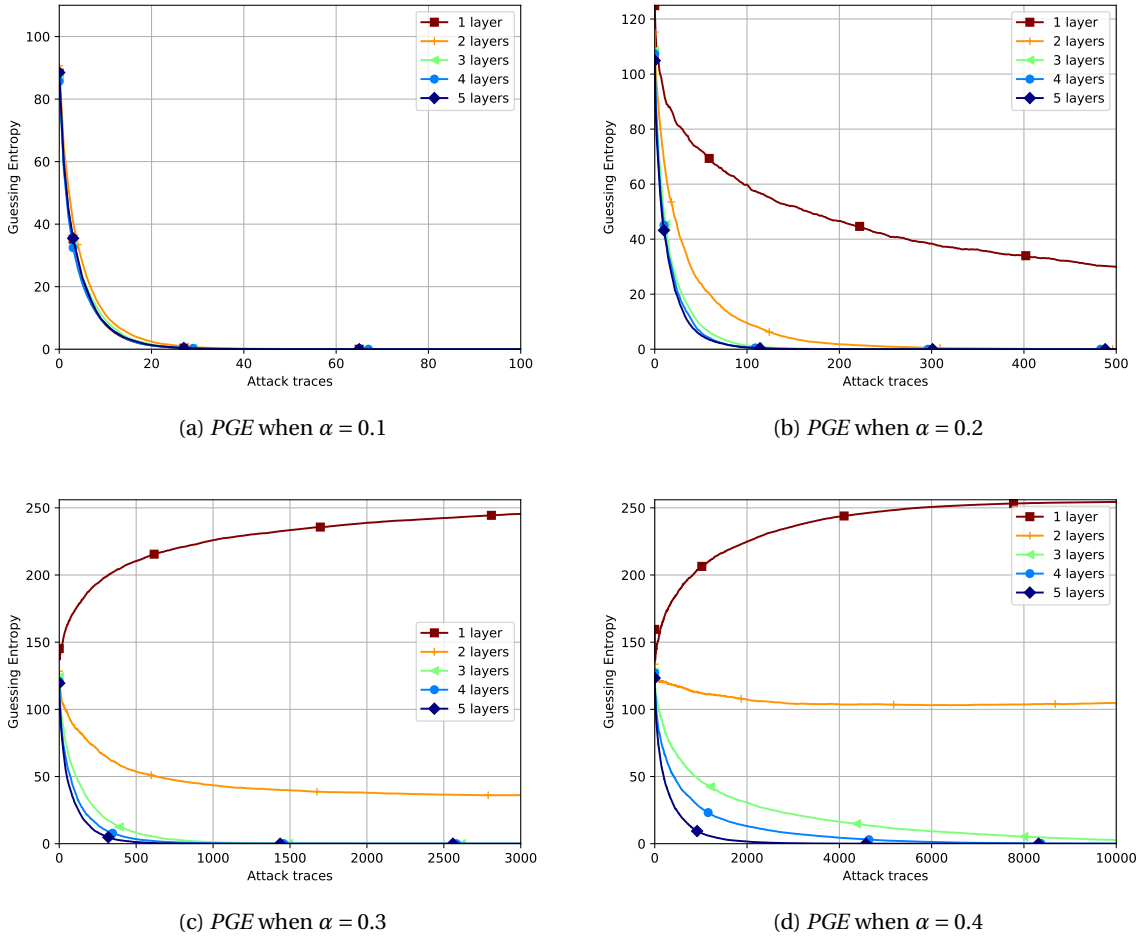


(d) *PGE* when $\alpha = 0.4$

Figure 5.8: *PGE* of $\boldsymbol{VGG}_{l,15}$ for adding various noise factors $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$, for various number of convolutional layers in a block on $ASCAD_D^{50}$

Noise (α)	**Conv layers in a block**				
	1	2	3	4	5
0.0	23	25	27	30	35
0.1	199	135	127	127	117
0.2	-	241	245	221	234
0.3	-	683	1 197	762	674
0.4	-	2 983	4 581	2 608	2 941

Table 5.12: Results for $\boldsymbol{VGG}_{l,15}$ with different noise levels for $ASCAD_D^{50}$



(a) *PGE* when $\alpha = 0.1$

(b) *PGE* when $\alpha = 0.2$

(c) *PGE* when $\alpha = 0.3$

(d) *PGE* when $\alpha = 0.4$

Figure 5.9: *PGE* of $\boldsymbol{VGG}_{l,15}$ for adding various noise factors $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$, for various number of convolutional layers in a block on $ASCAD_D^{100}$

Noise (α)	**Conv layers in a block**				
	1	2	3	4	5
0.0	23	29	22	23	29
0.1	184	205	150	168	196
0.2	-	856	283	238	208
0.3	-	-	3 386	2 694	1 361
0.4	-	-	-	9 929	6 231

Table 5.13: Results for $\boldsymbol{VGG}_{l,15}$ with different noise levels for $ASCAD_D^{100}$

the results of $\boldsymbol{VGG}_{l,15}$ using the datasets *RD*, $ASCAD_D^{50}$, and $ASCAD_D^{100}$. In only one setting increasing the number of layers did not result in better performance. For all other experiments, we have shown by using either, traces with and without noise, that increasing the depth of a network makes the networks more robust and decreases the *CGE*. For the results where the *CGE* is already low, we see that the deeper networks are more robust to noise. Furthermore, we have noticed that as the length of the artificial delay increases (thus datasets $ASCAD_D^{50}$ and $ASCAD_D^{100}$), deeper networks perform better on noisy traces.

5.3.3. Experimental Results Kernels and Convolutional Layers

So far we have analyzed the influence of the kernel size using a fixed number of convolutional layers in a block, and the influence of the number of convolutional layers in a block by fixing the kernel size. In this section, we will have neither the kernel size nor the number of convolutional layers fixed and will vary both. By doing so, we aim to gain an understanding of the relation of kernel size and number of convolutional layers in a block and provide us with enough information to answer **RQ 4.**.

As we will vary the kernel size and number of convolutional layers in a block we will gain more understanding of the influence of hyperparameters. Therefore, these findings help us to provide a more detailed answer for **RQ 2.** and **RQ 3.**.

For the experiments performed in this section, we use the same settings as discussed in the previous section (listed in Table 5.1) and start the experiments with $\boldsymbol{VGG}_{l,k}$, the values for l and k are listed in Table 5.14. Again, we consider the three datasets where the traces are obtained from an AES implementation with a random delay countermeasure, namely *RD*, $ASCAD_D^{50}$, and $ASCAD_D^{100}$. In this section, we depict the results in a heatmap. In these figures, the vertical axis represents the kernel size and the horizontal axis the number of layers from the CNN. Furthermore, on the right side of these figures, a legend is show2n that highlights which metric is used in the figure.

#Layers	Kernel sizes
1	100, 50, 25, 20, 15, 10, 7, 5, 3
2	100, 50, 25, 20, 15, 10, 7, 5, 3
3	50, 25, 20, 15, 10, 7 , 5, 3
4	25, 20, 15, 10, 7, 5
5	20, 15, 10, 7, 5, 3

Table 5.14: Varying architecture settings

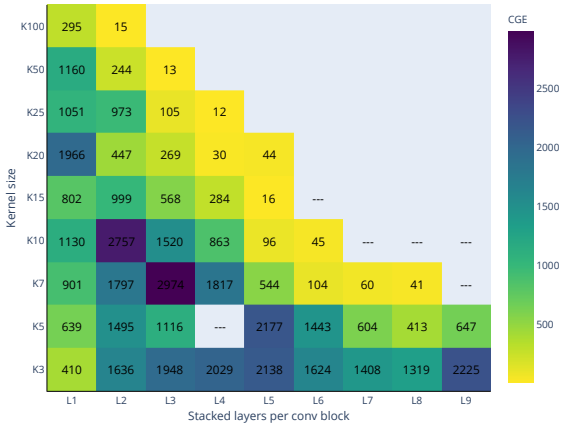
#Layers	Kernel sizes
6	15, 10, 7, 5, 3
7	10, 7, 5, 3
8	10, 7, 5, 3
9	10, 7, 5, 3

Table 5.15: Additional architecture settings for *RD*

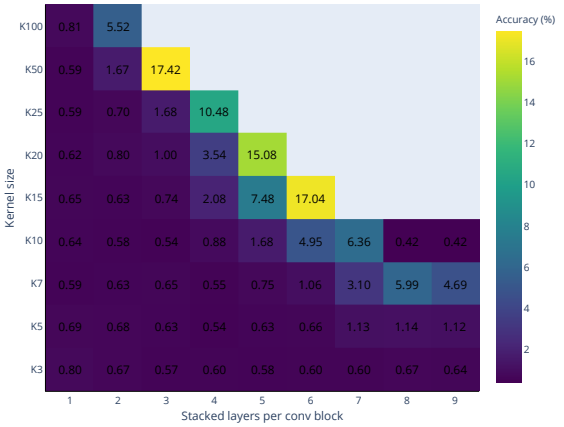
Dataset *RD*

In Figure 5.10a we depict the *CGE* of the experiments in a heatmap. From a kernel size perspective, the best performing models are the ones that have the largest kernel size in each layer. From the number of convolutional layers perspective, our observations are a bit different. For a kernel size of three and five, we see no significant improvement in the *CGE* when adding more layers. However, with a kernel size bigger than five we see that adding layers decreases the *CGE* and thus increases the attack efficiency.

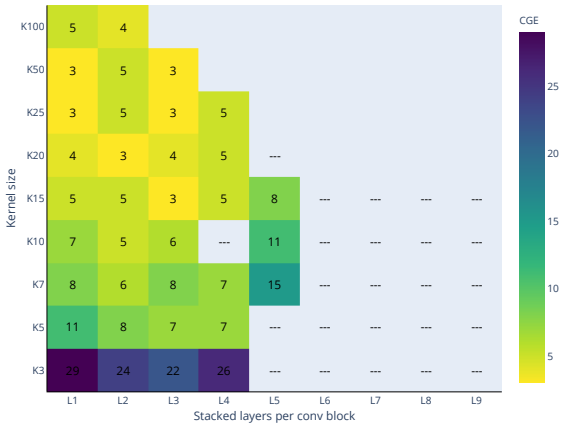
When considering more than five layers we see that the highest kernel sizes are filled with blanks. This means that in at least one of the five folds the model was not able to successfully recover the key. In these cases, there was indeed one model that was not able to extract the key successfully. These models were not able to recover the key because they were "stuck" learning, meaning that the validation loss does not change over the epochs. We believe this problem occurs because the models experience the vanishing gradient problem, which occurs because the architecture consists of a combination of too many layers and too-large kernel size. Thus by adding more layers we see that the larger kernel size can not extract the key. For example, for $l = 6$ and $k = 15$ this is the first kernel size that is not able to recover the key for this number of layers. If we add an additional layer, thus $l = 7$, we see that the highest kernel size that can recover the key is $k = 7$. For $l = 9$, we see that the highest kernel size that can recover the key is $k = 5$. Another outlier we see in this figure, is when $l = 4$ and $k = 5$, here, a similar situation occurs as the just described problems, there was one model which was not able to recover the key. It is however not clear why this happens specifically for these values. For the last outlier we discuss, we look at the *CGE* and accuracy of $\boldsymbol{VGG}_{6,15}$ in Figure 5.10a and Figure 5.10a. For this setting, the accuracy is highest of all networks, but the *CGE* shows that not all models were able to



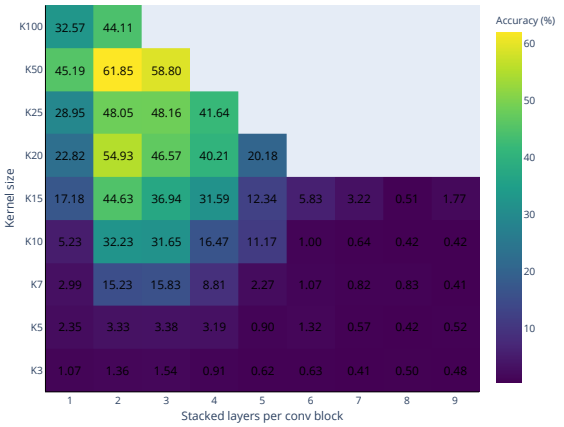
(a) *CGE* with $\lambda = 0.0$



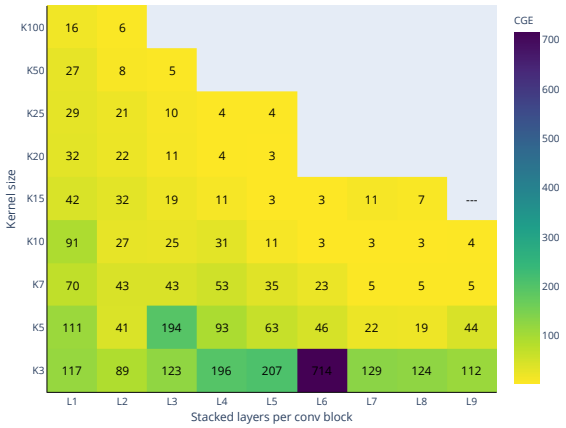
(b) Accuracy with $\lambda = 0.0$



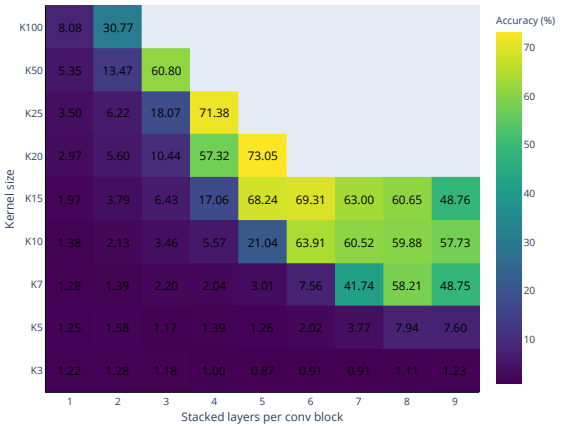
(c) *CGE* with $\lambda = 0.05$



(d) Accuracy with $\lambda = 0.05$



(e) *CGE* with $\lambda = 0.005$



(f) Accuracy with $\lambda = 0.005$

Figure 5.10: *CGE* and accuracy of the $\boldsymbol{VGG}_{l,k}$ trained with L2-penalties $\lambda \in \{0, 0.05, 0.005\}$ on the dataset *RD*.

recover the key. Further analysis showed that one network did not converge and was therefore not able to recover the key.

In the previous sections, we have experimented with different values for an L2-penalty. For a example when $\lambda = 0.05$, we observed that increasing the number of layers negatively impacts the *CGE*, while for $\lambda = 0.005$ we observed the opposite. Therefore we are interested in the influence of λ, and thus perform some experiments on the influence of L2-regularization in these settings. We apply the same L2-penalties as the previous section and depict the *CGE* of an L2-penalty of 0.05 and 0.005 in Figure 5.10c and Figure 5.10e respectively.

When $\lambda = 0.05$ and $l > 5$ we see that no kernel size is able to retrieve the key. This indicates that increasing the number of layers does not help in achieving a better classifier. This is consistent with what we have observed in the discussion about the influence of the number of layers. Furthermore, we observe that the *CGE* is similar when $k > 10$ and $1 \leq l \leq 4$. When analyzing the accuracy over the entire attack set we notice that the best performing classifiers are centered around $l = 2$ and $k = 50$. Increasing or decreasing the kernel size or the number of layers results in decreasingly performing classifiers.

If we consider $\lambda = 0.005$ we see that all models can retrieve the key. Additionally, we observe that an increase in the kernel size results in a decrease in the *CGE*, and an increase in the number of layers depicts similar behavior. However, when $k = 3$ we see no improvement when increasing the number of layers. Furthermore, we observe that the best *CGE* is achieved by several combinations of high kernel sizes and many layers. By analyzing the accuracy the difference between the models becomes more clear. The best models achieve around 70% accuracy and are the ones where $l = 4 \wedge k = 25$, and $l = 5 \wedge k = 20$. Notice that these kernel sizes are the highest kernel sizes we have tested in the respective number of layers. The models might even perform better by increasing the kernel size although this might harden the learning process.

In the previous sections, we have used noisy traces when the guessing entropy is similar for the tested hyperparameters. We have also done this for the settings discussed in this section. However, we do not discuss them here, but show the results in Figure A.1 and Figure A.2 in Appendix A for interested readers.

Datasets $ASCAD_D^{50}$ and $ASCAD_D^{100}$

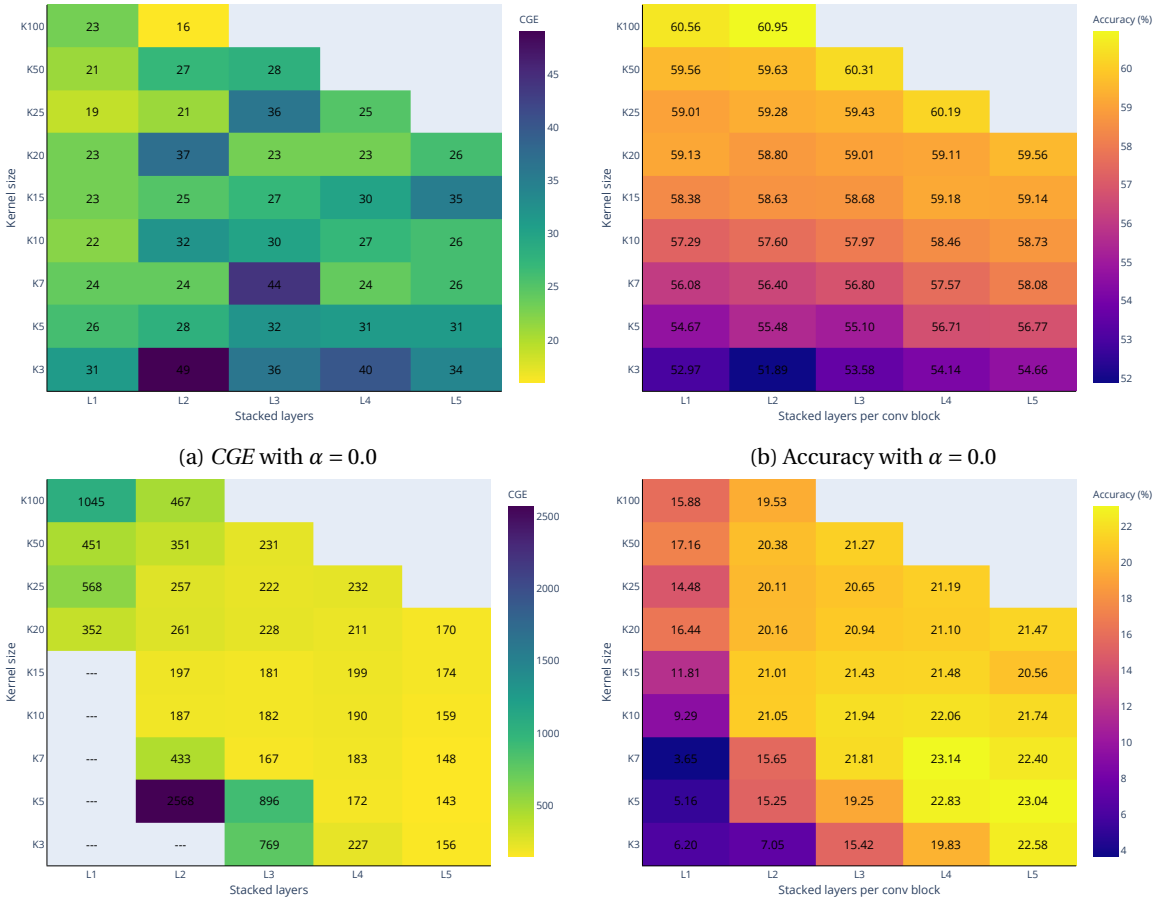
Here we consider the two datasets of the ASCAD database, $ASCAD_D^{50}$, and $ASCAD_D^{100}$. In Figure 5.11 we depict the experimental results, the *CGE*, and the accuracy of the architectures. Figure 5.11a depicts the *CGE* using the attack traces without any noise. In this figure, we observe similar *CGE* for the architectures. However, it seems that increasing the kernel size (for all values of l) results in better-performing models. The accuracy heatmap, depicted in Figure 5.11b, shows similar behavior, except that increasing the number of layers increases the performance of the models as well. To observe a more significant difference in the *CGE*, we resolve to experiments with the noisy attack traces.

The noise factor used for the experiments are $\alpha \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$, however, here we only show the results of the noise factor $\alpha \in \{0.0, 0.2, 0.3, 0.4\}$ since these are the most interesting. First, we will discuss the influence of the kernel size. As the noise increases, we observe architectures with larger kernel sizes can recover the key, while the lower ones can not. For example, for $\alpha = 0.2 \wedge l = 1$, a kernel size of 20 and larger are the only networks that can recover the key. When analyzing the influence of the depth of the network, we observe that deeper networks perform better. In Figure 5.11e this can be observed, when we look at the kernel size five we see that only the networks that have four and five layers can recover the key. These observations are in line with the previous sections and show that our conclusions were correct.

Figure 5.11e and Figure 5.11g, where $\alpha = 0.3$ and $\alpha = 0.4$ respectively, show that the best performing architectures are centered around $\boldsymbol{VGG}_{5,5}$. Furthermore, we observe that as we increase the network depth', from the networks that can recover the key, the ones where the kernel size is the lowest perform the best in *CGE* and accuracy. For example, when $\alpha = 0.4$ and $l = 2$ we observe that $15 \leq k \leq 50$ successfully recover the key, and $k = 15$ has the lowest *CGE*. Similar behavior can be observed for different noise levels and the number of layers. Note that, here we do not claim that lower kernel sizes perform better.

For $ASCAD_D^{100}$ we depict the experimental results in Figure 5.12 for the noise factors $\alpha \in \{0.0, 0.2, 0.3, 0.4\}$. The observations of the influence of the kernel size and depth of the network are similar to the observations of $ASCAD_D^{50}$. However, we still discuss these findings because we are interested in the difference between the length of the random delay. The major difference we observe is the importance of the architecture's depth. When the length of the random delay increases, the deeper architectures perform better. This becomes clear if we look at the results for $ASCAD_D^{50}$ and $ASCAD_D^{100}$ with $\alpha = 0.3$ in Figure 5.11e and Figure 5.12e. Here, we see that the shallow networks are able to recover the key for the dataset $ASCAD_D^{50}$ but not for the dataset $ASCAD_D^{100}$.

In general for both datasets, we conclude that both the kernel size and depth of a CNN are important for efficient attacks. A small kernel and shallow networks does not perform well. By increasing one of these the attack efficiency increases as well. However, by using a large kernel size, around 20, we observe that all number of layers perform efficient attacks. It is thus suggested to use at least a kernel size of 20 for SCAs where a random delay countermeasure is present, additionally, at least two convolutional layers in a block have to be used.



(a) *CGE* with $\alpha = 0.0$

(b) Accuracy with $\alpha = 0.0$

(c) *CGE* with $\alpha = 0.2$

(d) Accuracy with $\alpha = 0.2$

Figure 5.11: *CGE* of the $\boldsymbol{VGG}_{l,k}$ trained on $ASCAD_D^{50}$ attacking noisy traces with noise factors $\alpha \in \{0.0, 0.2, 0.3, 0.4\}$ (cont. on next page)

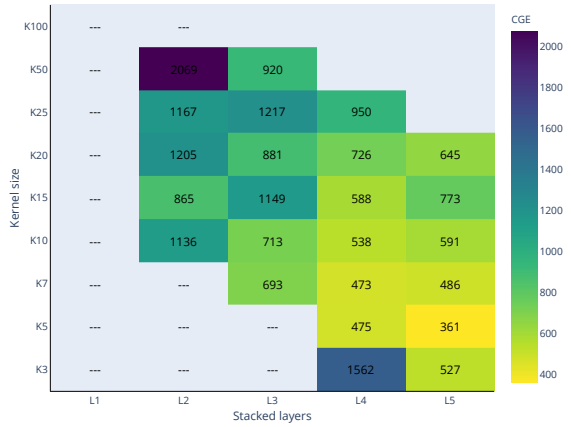
5.4. The Masking Countermeasure

In this section we analyze the influence of the kernel size and depth of the network for AES-implementations with a masking countermeasure. We will use only one dataset for our experiments, namely $ASCAD_M$. Furthermore, we perform the same experiments as the previous sections, thus will use $\boldsymbol{VGG}_{l,k}$, to perform our analysis. However, in this section, we will only give an overview of the guessing entropy and accuracy using the heatmaps as shown in subsection 5.3.3.

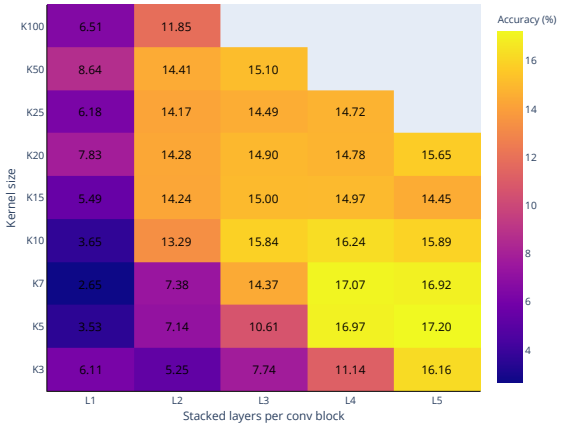
5.4.1. Experimental Results

The experiments performed in this section are conducted with the architecture $\boldsymbol{VGG}_{l,k}$. Like the experiments for the random delay countermeasure, we vary the depth and kernel size of the architecture. To be precise, the values for l and k are listed in Table 5.14. The results are depicted in Figure 5.13, in which we depict the guessing entropy after 10 000 traces and the accuracy. We list the guessing entropy after 10 000 traces because the learned models do not perform well. If they can retrieve the key, then they require around 9 900 traces, and the difference in *CGE* is minimal. Figure 5.13a shows that deeper networks have more trouble with recovering the key. Analyzing the kernel size, we observe that an increase of the kernel size results in a better guessing entropy after the 10 000 traces, and some settings can recover the key consistently. However, if we increase the kernel size too much, we see that it performs significantly worse than all other settings. For example, we observe this in the settings where $l = 3 \wedge k = 25$, and $l = 4 \wedge k \geq 20$. We think this occurs because the kernel size is too high for the number of layers, and we again struggle with vanishing gradients.

Considering the accuracy, shown in Figure 5.13b, we observe somewhat different behavior. Similar to the guessing entropy heatmap, we observe that adding more layers decreases the accuracy. However, for more than two layers in a convolutional block we observe that the accuracy decreases, while in general the guessing

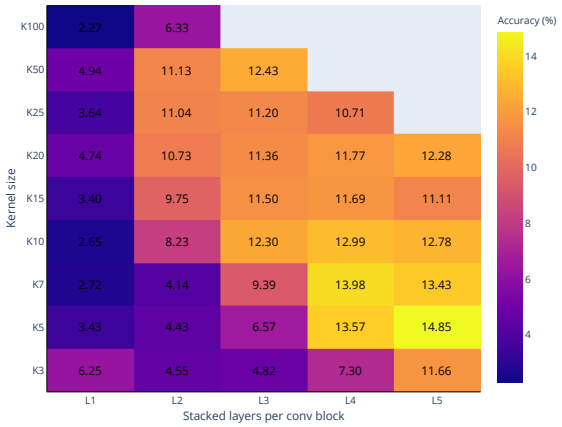


(e) *CGE* with $\alpha = 0.3$



(f) Accuracy with $\alpha = 0.3$

(g) *CGE* with $\alpha = 0.4$



(h) Accuracy with $\alpha = 0.4$

Figure 5.11: *CGE* of the $\boldsymbol{VGG}_{l,k}$ trained on $ASCAD_D^{50}$ attacking noisy traces with noise factors $\alpha \in \{0.0, 0.2, 0.3, 0.4\}$ (cont)

entropy improves. This could be explained by Picek et al. [41] which showed that accuracy is not best metric in SCA.

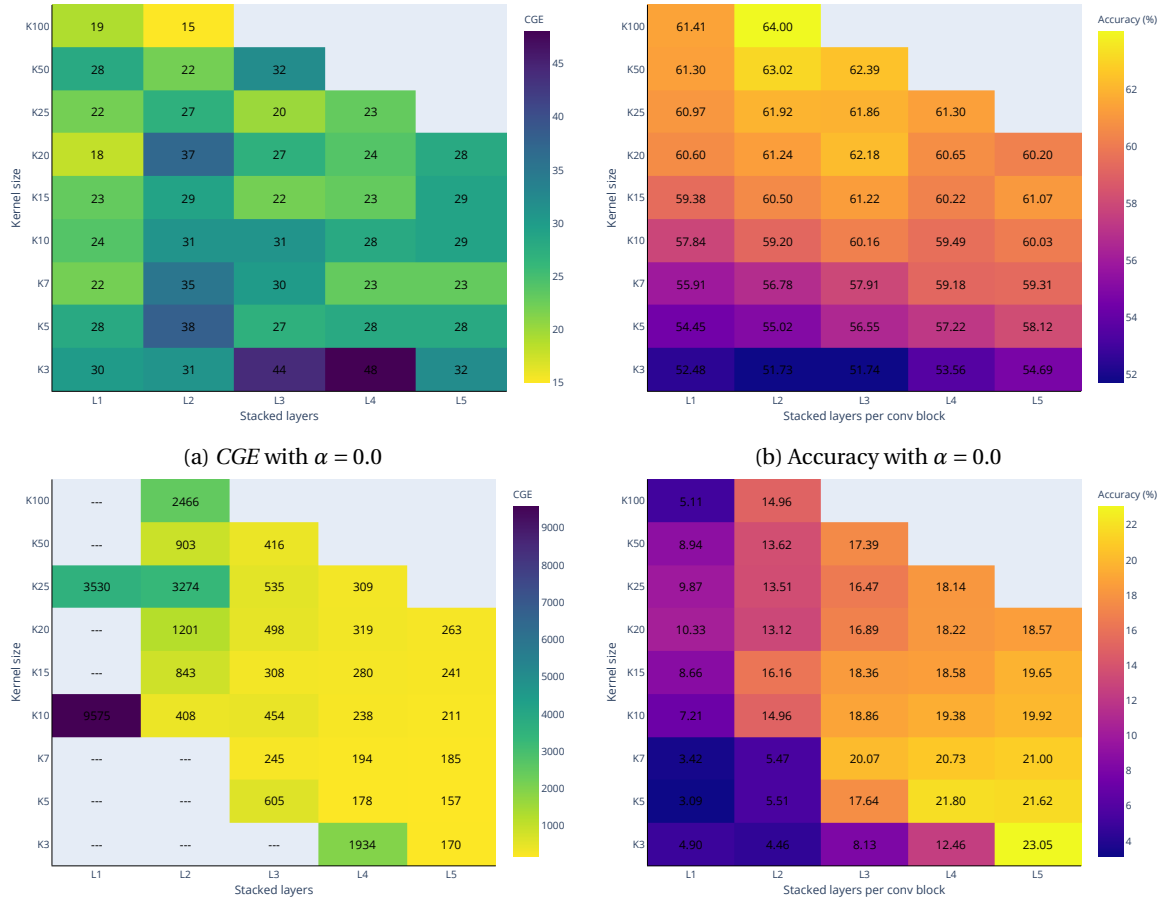
The performance of the CNNs is far from the performance of $\boldsymbol{Dense}_{\boldsymbol{BN}}$ (introduced in the previous chapter, see Figure 4.9). The latter network is around nine times as efficient than the best-performing CNN. From our results, it thus seems that CNNs are not suitable for attacking a masked implementation, while an MLP network. Intuitively, we can explain these results by comparing a high-order attack with an MLP network. In such an attack, two PoIs are selected and multiply them to retrieve the intermediate value. In an MLP network, similar behavior occurs. Since an MLP network is a fully connected network, each feature of a trace is combined with all features, however, the network learns the weights it should assign to these features. Thus if the network assigns proper weights to features, an MLP network could perform a higher-order attack.

5.5. Conclusions

In this chapter, we have discussed the influence of the kernel size and depth of an architecture for the two main countermeasures: random delay and masking. Furthermore, we discussed the consideration between the kernel size and the number of layers. First, we will discuss our experimental results for the hiding countermeasure, and then the results for the masking countermeasure.

In general, we observe that increasing the kernel size is highly effective for SCA. The difference in *CGE* can be significant, but we have also shown that similar *CGE* can be achieved by decreasing the proneness to overfit. When the *CGE* was already low, we have shown by adding noise to the attack traces that the larger kernel sizes result in better classifiers.

For the depth of the architecture we observed similar behavior; increasing the depth resulted in better classifiers for SCAs. However, unlimitedly increasing the depth of a network does not increase attack effi-



(a) *CGE* with $\alpha = 0.0$

(b) Accuracy with $\alpha = 0.0$

(c) *CGE* with $\alpha = 0.2$

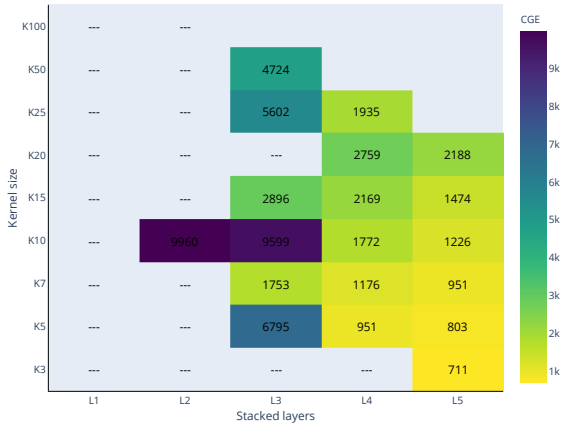
(d) Accuracy with $\alpha = 0.2$

Figure 5.12: *CGE* of the $\boldsymbol{VGG}_{l,k}$ trained on $ASCAD_D^{100}$ attacking noisy traces with noise factors $\alpha \in \{0.0, 0.2, 0.3, 0.4\}$ (cont. on next page)

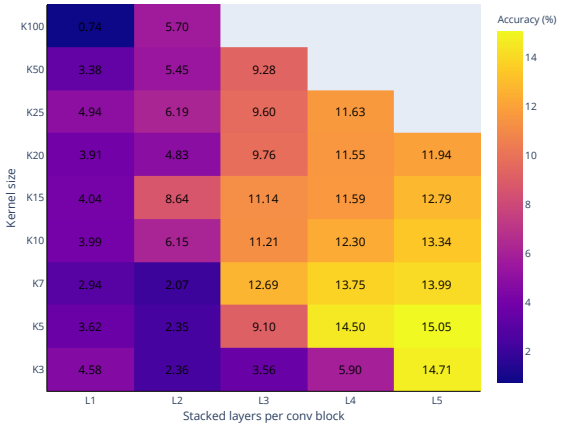
ciency. On the contrary, eventually, the classifiers fail to recover the key because they underfit severely. Similarly, as for the kernel size, we have added noise to the attack traces if the *CGE* was similar for all architectures. These results showed that an increase in the number of convolutional layers in a block results in better *CGE*. However, there was one exception, which was on *RD* trained using an L2-penalty $\lambda = 0.05$. With the analysis of the kernel size versus depth of the network, it seems that this occurs because of L2-regularization. It seems as if L2-regularization allows us to shift the center of best-performing architectures such that fewer layers are required to achieve the best performing architectures. This becomes more clear when analyzing the accuracy of these settings.

A comparison between the kernel size and depth of an architecture is hard to perform for both datasets since the results were somewhat different. However, we have observed that the best results are achieved with $2 \leq l \leq 4$, and $15 \leq k \leq 50$.

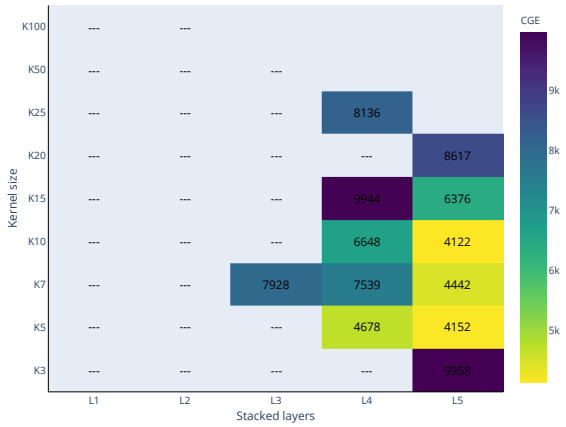
The results for measurements obtained from a masked implementation are different. All trained networks were not efficient and mostly failed to retrieve the key. For this setting, adding more layers decreases the attack efficiency and increasing the kernel size increases the attack efficiency. However, the gains of increasing the kernel size are marginal. Additionally, we argue that MLP networks perform better than CNNs because they are in some sense similar to a higher-order attack.



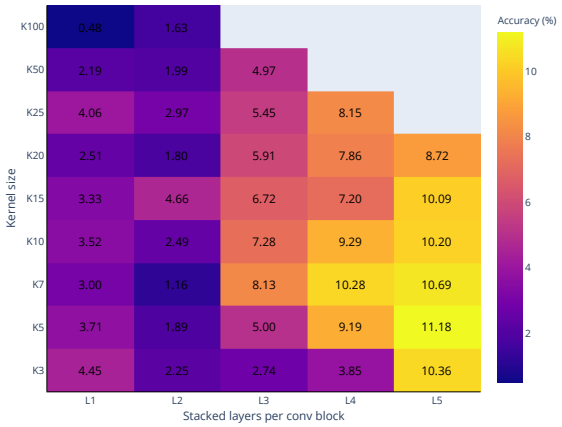
(e) *CGE* with $\alpha = 0.3$



(f) Accuracy with $\alpha = 0.3$



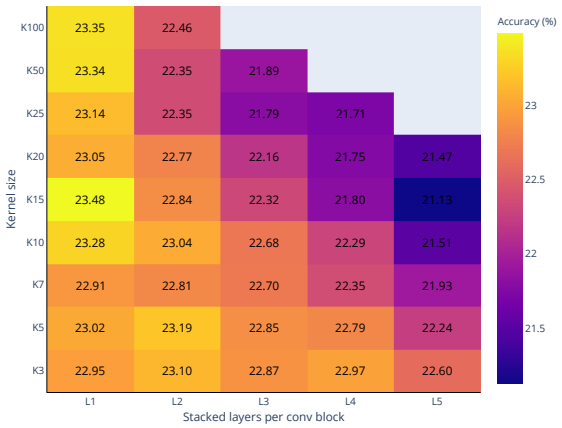
(g) *CGE* with $\alpha = 0.4$



(h) Accuracy with $\alpha = 0.4$

Figure 5.12: *CGE* of the $\boldsymbol{VGG}_{l,k}$ trained on $ASCAD_D^{100}$ attacking noisy traces with noise factors $\alpha \in \{0.0, 0.2, 0.3, 0.4\}$ (cont.)

(a) GE after 10 000 traces



(b) Accuracy

Figure 5.13: Guessing entropy and accuracy for $\boldsymbol{VGG}_{l,l}$ on $ASCAD_M$

6

Portability

In the SCA domain, most literature do not perform the attacks in a realistic setting. They are unrealistic because the profiling and attack traces are obtained from the same device rather than two distinct devices. Recent work has shown that the SCAs performed in such an unrealistic setting greatly overestimates the efficiency of an attack [4]. Thus, this means SCAs in realistic settings are harder to perform.

In this chapter, we will examine a more realistic SCA setting, in which the collection of profiling and attack traces is different as considered by most work. Specifically, the traces originate from the same device, but the probe position has been changed in between the measurements of the profiling and attack traces. The repositioning of the probe causes a difference between the profiling and attack traces. Although this is not the most realistic setting, in which two distinct devices are used for the measurements, we believe this is an interesting setting that could provide meaningful insights for more realistic settings.

The outline of this chapter is as follows. First, we discuss the dataset that we use for our experiments in more detail, followed by a typical SCA using the traces from this dataset. Since the portability setting causes the SCA to be ineffective, we analyze the difference between the profiling and attack traces in the following section. Additionally, we propose a method that allows us to eliminate the difference between the traces. In the next section, we show the proposed method enables the SCA to be successful. In the following sections, we analyze the influence of the training size, and the number of epochs on the guessing entropy when using the proposed method. Finally, we provide a conclusion of the observed results.

6.1. Dataset

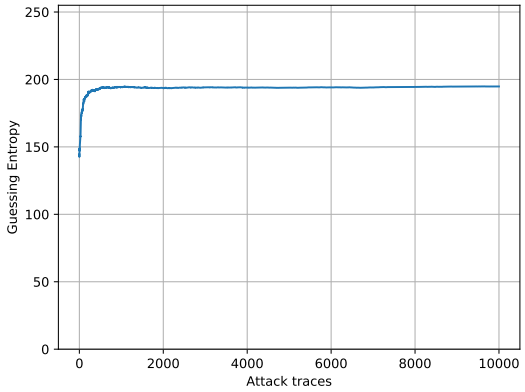
The dataset used for the experiments is different from the datasets used in the previous chapters. The key difference is that the probe has been repositioned in between the measurements of the profiling and attack traces. The repositioning of the probe is in the order of millimeters and causes a difference in the profiling and attack traces. Additionally, the datasets used in the previous chapters use the same key for profiling and attacking. The dataset used in this chapter has randomized plaintexts and keys for the profiling measurements, while a fixed key and randomized plaintexts have been used for the attack measurements. In subsection 2.6.4, we provide more details about the origin and technical aspects of the measurements.

6.2. Portability Setting

To analyze if this portability setting influences the efficiency of an SCA, we perform an SCA on this dataset. Moreover, for the attack, we consider the MLP network $\boldsymbol{MLP_{best}}$ and train the network for 75 epochs with 40 000 profiling traces, batch size of 256, a learning rate of 10^{-4}, and intermediate value as leakage model. For the attack phase, we have used 10 000 attack traces.

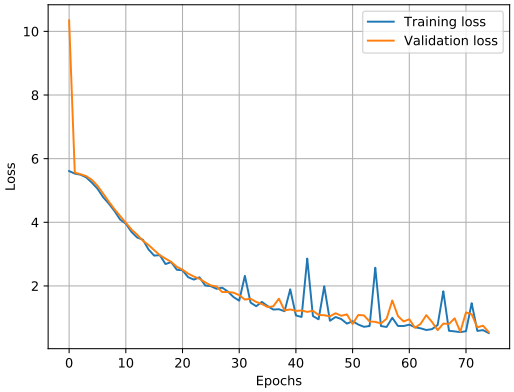
In Figure 6.1 we depict the guessing entropy, validation loss, and validation accuracy. From the guessing entropy plot in Figure 6.1a, we observe the attack is not successful. The guessing entropy does not converge to zero but is constant around 198, which could mean that increasing the attack set for the attack is not beneficial.

If we only analyze the resulting guessing entropy, we could conclude that the network's architecture is suitable, and incorrect hyperparameters have been used. However, when analyzing the validation loss and accuracy we see this hypothesis is not likely. From the validation loss and accuracy we observe no major



(a) Guessing entropy

(b) Validation accuracy during training



(c) Validation loss during training

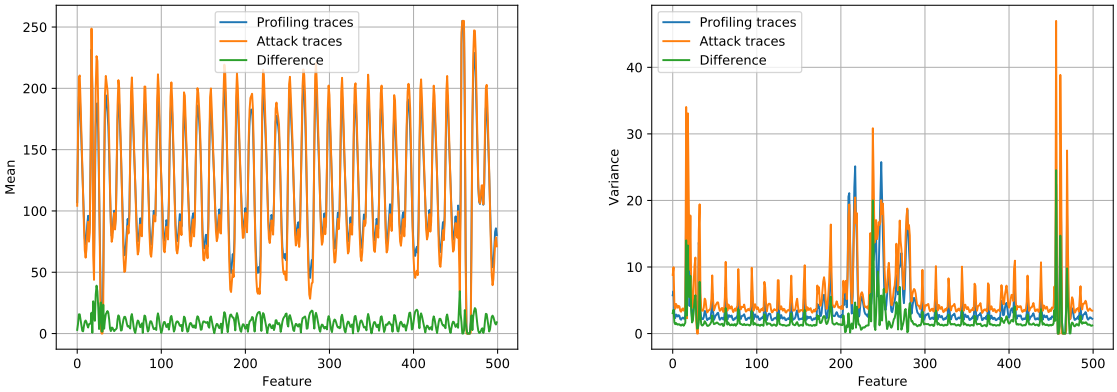
Figure 6.1: Guessing entropy, validation accuracy, and validation loss when attacking a portability setting.

overfitting, which shows the validation loss and accuracy are close to the training loss and accuracy. This thus means the network can accurately predict the profiling traces. However, when considering the attack traces we observe the network is not able to recover the key, meaning the network's predictions are incorrect. We believe the difference between the network's ability to predict the profiling and attack traces is caused by the repositioning of the probe. In the following section, we will analyze the difference between the profiling traces and attack traces.

6.2.1. Traces Analysis

As we believe the repositioning of the probe has a significant influence on the attack success, we analyze the difference between the profiling and attack traces. By doing so we aim to observe characteristics that allow us to develop a successful attack. To analyze the traces we determine the mean and variance of each feature of the profiling and attack traces. In Figure 6.2 we depict the mean and variance of profiling and attack traces, additionally, we plot the absolute difference between the mean and variance of the features. From the plot of the means in Figure 6.2a, we observe that the profiling and attack traces have similar behavior but the strength of the signal differs. The largest difference in the signal's strength observed is around 40, which is about one-sixth of the largest amplitude of the attack traces. In Figure 6.2b we observe similar differences, the features' variance of the profiling and attack traces are similar, but the variance differs. These differences could explain why the attack performed in the previous section has not been successful.

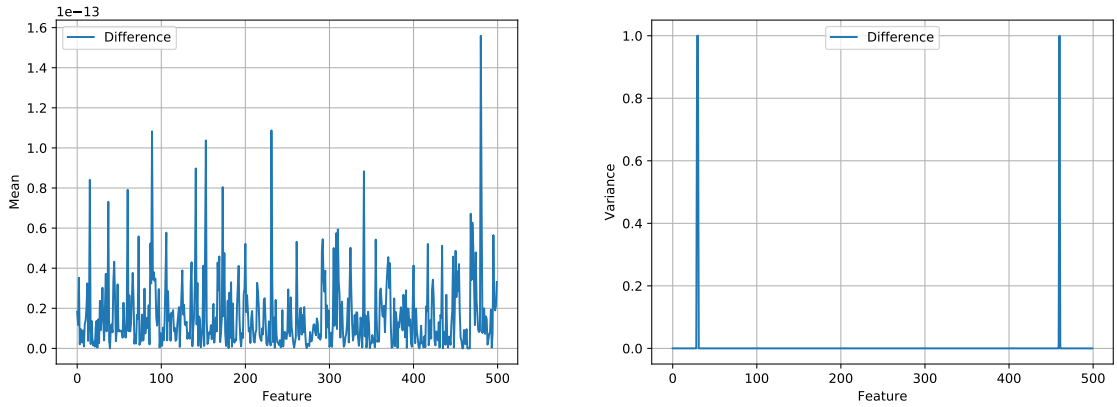
The difference between the profiling and attack traces seem to be caused by the difference in mean. Therefore we suggest to normalize the profiling and attack traces separately. By doing so we aim to minimize the difference between the profiling and attack traces. In Figure 6.3 we depict the difference of the mean and variance between the datasets when using this method. For both the features' mean and variance



(a) Mean of the traces' features

(b) Variance of the traces' features

Figure 6.2: Mean and variance of the traces' features



(a) Mean difference of the traces' features

(b) Variance difference of the traces' features

Figure 6.3: Difference of means and variances of the traces' features when separately normalizing the datasets.

the difference has been reduced significantly. The difference in mean is almost equal to zero for all features, for the variance there is no difference except for two features. Thus, by using this method the profiling and attack traces become similar, which might make a deep learning SCA possible. In the following section, we perform an SCA using this method.

6.3. Normalized Attack

For the attack we use identical hyperparameters as discussed in section 6.2, however, for this attack we use the proposed normalization method to eliminate the difference of the profiling and attack traces. Thus, before the profiling phase, we normalize 40 000 profiling traces and 10 000 attack traces separately. In Figure 6.4 we depict the guessing entropy using the proposed method, from this figure we observe the attack is extremely successful. Typically, we require at most two traces to recover the key, although when using a single trace we are almost certain the correct intermediate value has been predicted, and thus the correct key. The accuracy of the attack traces is around 99.8%, which explains the resulting guessing entropy.

The attack success shows the proposed method is efficient for settings where the probe has been moved in between measurements of the profiling and attack phase. However, arguing a single trace is required for the attack is not correct since 10 000 traces have been used to normalize the attack traces. To create a correct analysis of the amount of traces required to recover the key we should normalize a subset of the attack traces and calculate the guessing entropy with the normalized subset. In the following section, we will discuss how to calculate the guessing entropy when using the proposed method. Since the method achieves efficient

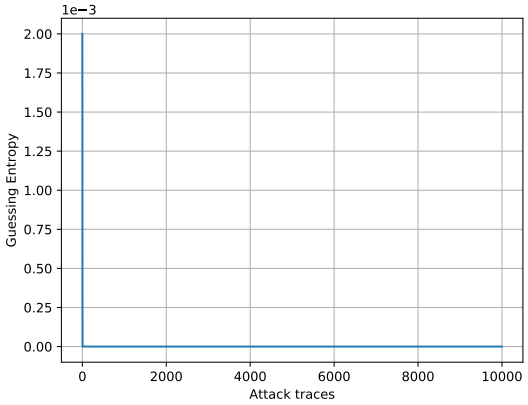


Figure 6.4: Guessing entropy when normalizing the profiling and attack traces separately.

SCAs in this portability setting, we are interested in the attack efficiency when considering different leakage models, amount of profiling traces, and epochs. To analyze the efficiency we first create a baseline, with which we compare the efficiency of various hyperparameters configurations.

6.3.1. Guessing Entropy Calculation

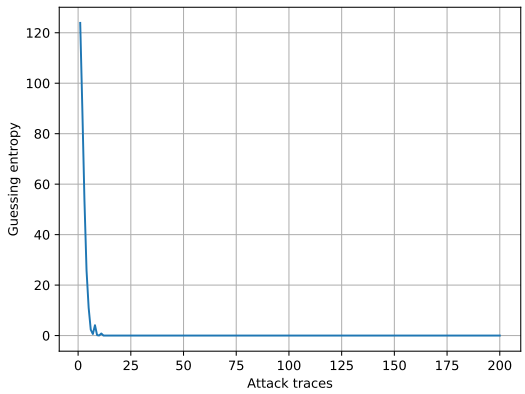
To avoid confusion between the guessing entropy as discussed in chapter 2 and this section, we denote the former as GE and the latter as NGE. The NGE for n attack traces is equal to the average of the GE of 100 subsets of n arbitrary traces from the attack set (for each subset we thus select n arbitrary traces). Thus, to create an accurate view of the GE, the NGE for a range of n attack traces has to be calculated. More specifically, we denote the entire attack set as T_a, and n as the number of traces used to perform an attack. To calculate the NGE for n traces we average the GE at trace n by performing the following algorithm 100 times.

First, randomly select n traces from the attack set T_a, we denote this subset as T_a^n. Then, normalize T_a^n using the traces in this subset, which we will denote as $\hat{T}_a^n$. Finally, using the learned neural network we generate predictions for $\hat{T}_a^n$ and use these to calculate the GE. For the remaining part of this chapter we refer to the NGE as guessing entropy.

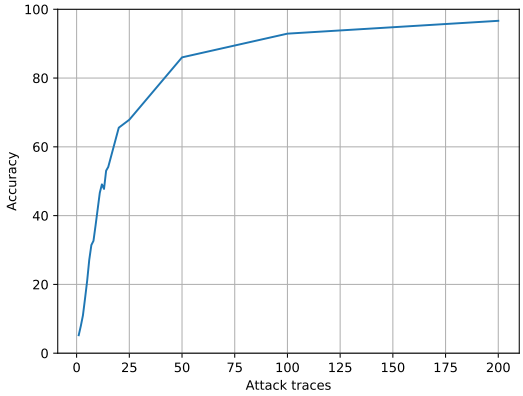
6.3.2. Baseline

To accurately observe the influence of the proposed normalization method when using the leakage models HW and ID, the amount of profiling traces, and epochs, we first develop a baseline for both leakage models. To create the baseline for both leakage models we require the optimal setting in which we obtain a "perfect" network. We define a "perfect" network as a network that can predict all the traces correctly with large confidence (i.e. the network assigns a high probability to the correct value). In the intermediate value model, such a "perfect" network allows us to recover the key using only a single trace, thus it enables us to determine the influence on the guessing entropy when using the proposed normalization method. For the HW model, the amount of attack traces required to recover the key with a "perfect" network depends on the key under attack. For an HW of 4 there are 70 possible keys, thus with a single trace it is not possible to recover the key, while for an HW of 0 and 8 it is possible since the only possible key is zero. We ignore this problem because it is inherent to the HW model and is also present in a real attack. Thus for intermediate value and HW model we establish a baseline by using a "perfect" network. These baselines provide us with insights on the amount of traces we minimally require to recover the key.

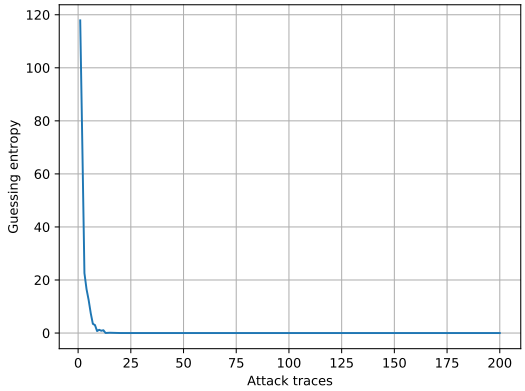
As shown in section 6.3 the network was able to predict almost all attack traces correctly. We have used this network to create the baseline when using intermediate value as leakage model since this network is near "perfect". When using HW as leakage model we have trained a network with identical settings as the "perfect" network when attacking the intermediate value. This network performs similar and predicts almost all attack traces correctly when using a large attack set. Figure 6.5a and Figure 6.5c depict the resulting guessing entropy of the "perfect" model for intermediate value and HW model respectively. For intermediate value model, we observe that around 13 traces are required to constantly reach a guessing entropy of zero, while for Hamming weight model we require 20 traces. In Figure 6.5b and Figure 6.5d we depict the accuracy for



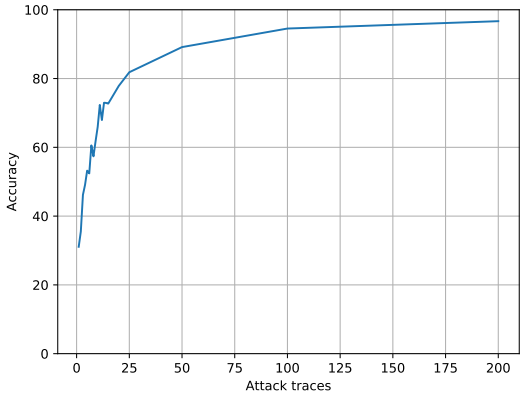
(a) Guessing entropy baseline for intermediate value model



(b) Accuracy baseline for intermediate value model



(c) Guessing entropy baseline for HW model



(d) Accuracy baseline for HW model

Figure 6.5: Guessing entropy and accuracy baseline for both leakage models

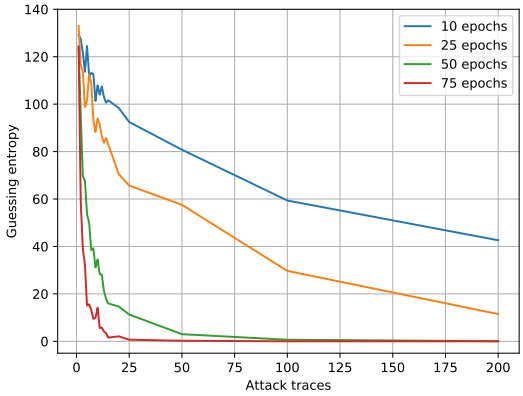
both leakage models. In these figures, we observe that increasing the amount of traces for the normalization increases accuracy. For both leakage models, we observe close to 100% accuracy when using 200 traces for the normalization.

6.3.3. Experimental Settings

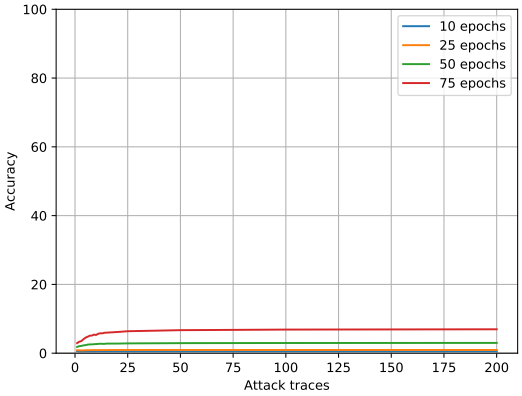
We aim to analyze the proposed method's influence when we do not have a "perfect" model. This allows us to analyze the attack efficiency in less favorable conditions. Therefore, we use configurations in which we reduce the amount of profiling traces, and reduce the number of epochs. These configurations allow us to learn networks that are not able to achieve around 100% accuracy. The hyperparameters we use for these experiments are listed in Table 6.1, each experiment is a combination of the values listed in this table. Furthermore, to calculate the guessing entropy of an experiment we use the following amount of attack traces $\{1,2,3,4,5,6,7,8,9,10,11,12,13,14,1520,25,50,100,200\}$.

Batch size	**LR**	**Training sizes**	**Epochs**	**Leakage models**
256	1e−4	$\{1\,000, 2\,000, 5\,000, 10\,000, 20\,000\}$	$\{10, 25, 50, 75\}$	{HW, ID}

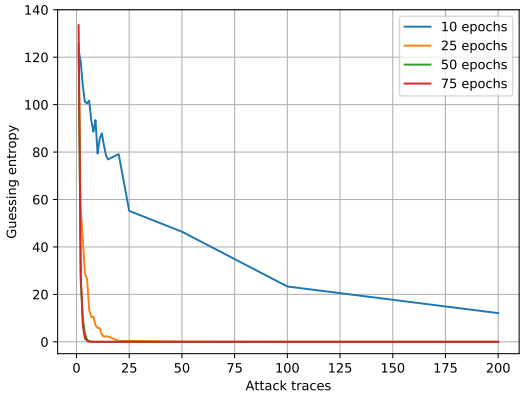
Table 6.1: Hyperparameters for imperfect model experiments



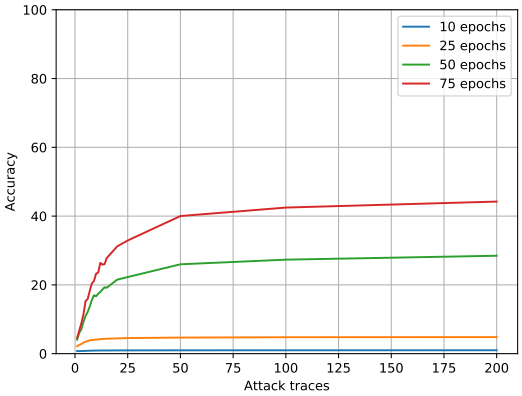
(a) GE using 1 000 profiling traces



(b) Accuracy using 1 000 profiling traces



(c) GE using 2 000 profiling traces



(d) Accuracy using 2 000 profiling traces

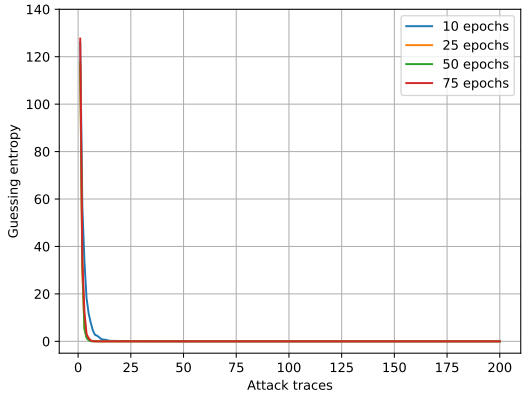
Figure 6.6: GE and accuracy when using different training sizes and intermediate value as leakage model. (cont. on next page)

6.3.4. Experimental Results

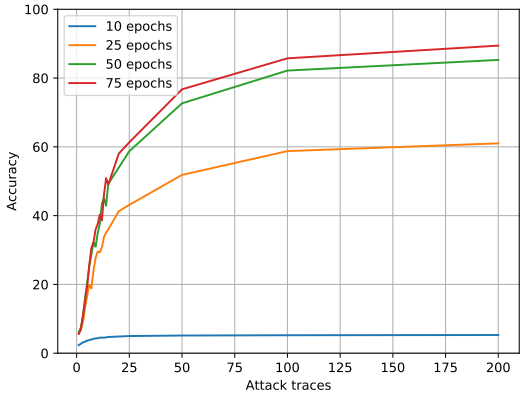
First we discuss the results for intermediate value followed by the results for HW. In Figure 6.6 and Figure 6.7 we depict the guessing entropy and accuracy of the networks using intermediate value as leakage model trained with the discussed hyperparameters. Note, we do not depict the results for networks (for both intermediate value and HW) which are trained with 20 000 training traces since the results for this configuration are similar to the baseline.

For the networks using intermediate value as leakage model and trained using 1 000 traces, we depict the results in Figure 6.6a, we observe an increase of epochs results in a decrease of guessing entropy and thus positively affects the efficiency of the attack. Although there is a significant difference in accuracy between the baseline and the network trained for 75 epochs, the difference in guessing entropy is small as well. For the network trained with 75 epochs, around 60 traces are required to recover the key. We observe similar behavior when the networks are trained with 2 000 traces, however, the influence of epochs on the guessing entropy is significantly decreased in comparison with the networks trained with 1 000 traces. When we analyze the accuracy, the influence of the number of epochs becomes visible; increasing the number of epochs increases accuracy. When comparing this setting to the baseline, we observe the networks trained with 50 epochs or more perform similar to the baseline. Increasing the training size further to 5 000, as depicted in Figure 6.6e, shows the influence of the number of epochs is minimal and only visible if the network is trained for 10 epochs. Finally, when training the networks with 10 000 traces the resulting guessing entropy is similar for all epochs. Networks trained with 10 epochs achieve to lowest accuracy but still achieve similar performance as the baseline.

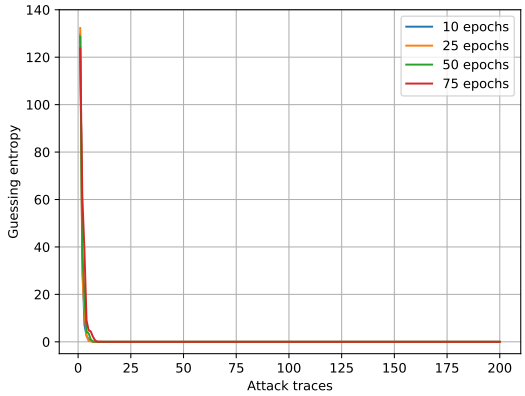
The results when using the HW model for the networks are depicted in Figure 6.7. The results are similar to the intermediate value model, however, HW model is more efficient than intermediate value when trained



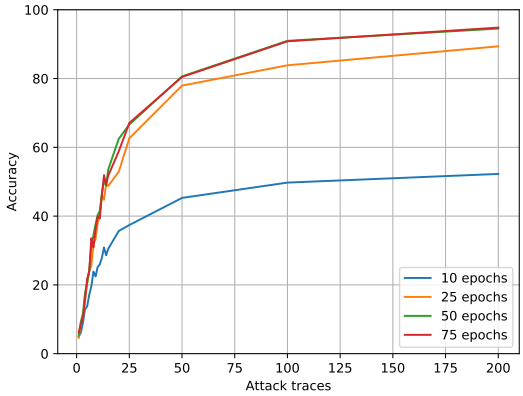
(e) GE using 5 000 profiling traces



(f) Accuracy using 5 000 profiling traces



(g) GE using 10 000 profiling traces



(h) Accuracy using 10 000 profiling traces

Figure 6.6: GE and accuracy when using different training sizes and intermediate value as leakage model. (cont.)

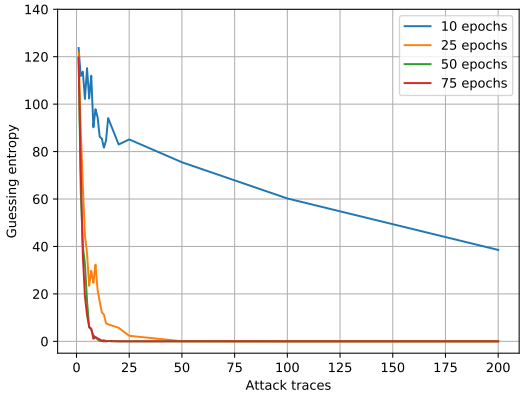
with less profiling traces and epochs. This is visible in Figure 6.7a, where we show the networks trained with 25 or more epochs can recover the key. When using intermediate value we require at least 50 epochs to recover the key. The results become similar to the baseline when the networks are trained with 5 000 traces. Like networks trained with intermediate value, we observe the accuracy is not close to the baseline, but the guessing entropy is almost identical to the baseline. Since these networks are as efficient as possible, increasing the training size even further does not improve the attack efficiency. The only difference that can be noticed is an increase in accuracy, as depicted in Figure 6.7e and Figure 6.7g.

In general, from the depicted results for both intermediate value and HW we make two observations: 1) training with more epochs increases the attack efficiency, and 2) increasing the training size the networks are more efficient and achieve better guessing entropy. This is similar to the settings we have previously seen for unprotected implementations.

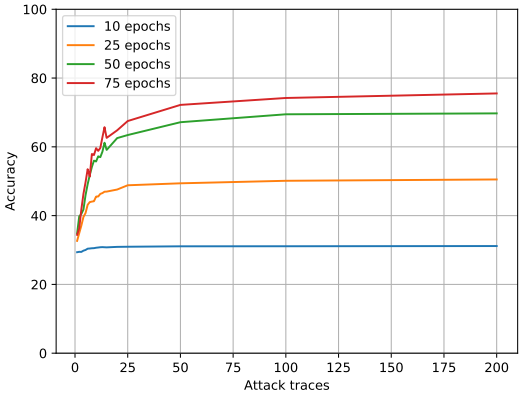
6.4. Conclusions

From our results we observe the probe position has a significant influence on an DL SCA, such that the attack is not possible. When analyzing the profiling and attack traces we observe differences between both the features' mean and variance. We observe that the key difference between the profiling and attack traces is the signal's strength which causes the difference in the traces. We believe this causes the typical DL SCA to be ineffective.

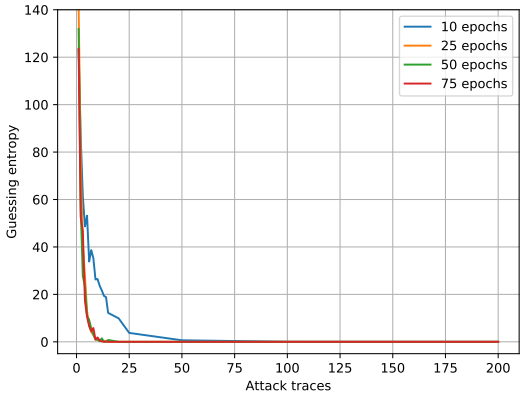
To counter the difference in the signal's strength we propose to normalize the profiling and attack traces individually. By doing so, the features of both datasets have zero mean and unit variance. If the datasets' traces have the same characteristics the attack should be successful. In this chapter we have shown that using



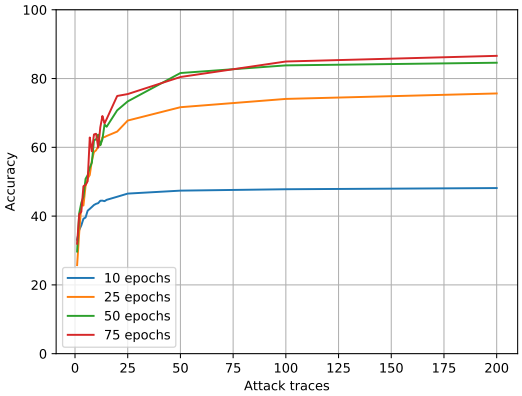
(a) GE using 1 000 profiling traces



(b) Accuracy using 1 000 profiling traces



(c) GE using 2 000 profiling traces



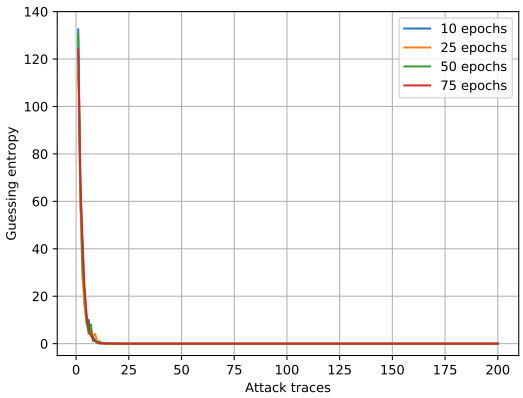
(d) Accuracy using 2 000 profiling traces

Figure 6.7: GE and accuracy when using different training sizes and HW as leakage model. (cont. on next page)

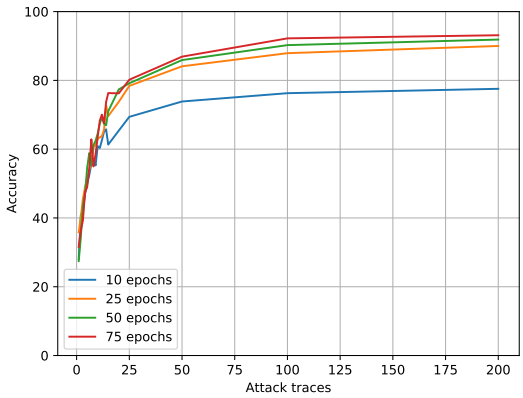
this method a successful SCA can be performed in such a portability setting. However, this method causes an incorrect image of the guessing entropy because the selected attack traces are required for the normalization. Therefore we introduce a method to correctly calculate the guessing entropy when using this normalization method.

Additionally, we have analyzed the efficiency of the normalization method in multiple settings. To do so we have first analyzed the efficiency of the "perfect" network for the leakage models intermediate value and hamming weight. For both settings we require at most 20 traces to recover the key. In the other settings, in which we vary the training size and amount of epochs, we observe that attack is more efficient when trained with more traces and for more epochs. This is in line with a typical SCA, which thus shows the proposed normalization method allows us to bridge the gap between the non-portability and portability setting discussed in this chapter.

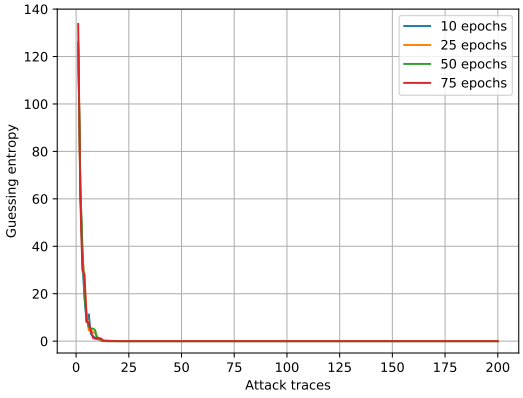
When discussing more realistic scenarios, in which the profiling and attack traces are obtained from different devices, the proposed method could make SCAs more efficient. In such a setting, it is near impossible to position the probe such that it is identically positioned for the copy and attack device. This difference will cause differences in the measured strength of the signal. Therefore, using the proposed normalized method would theoretically improve SCA efficiency in a portability setting, but future work is required to confirm this.



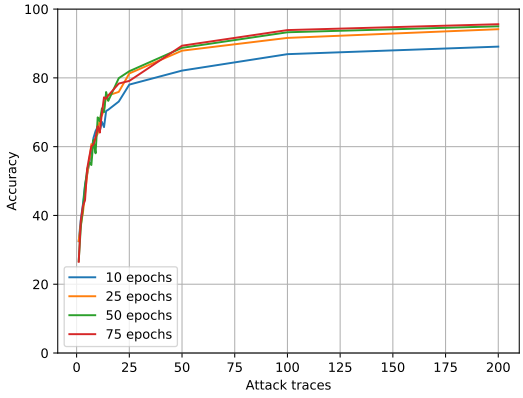
(e) GE using 5 000 profiling traces



(f) Accuracy using 5 000 profiling traces



(g) GE using 10 000 profiling traces



(h) Accuracy using 10 000 profiling traces

Figure 6.7: GE and accuracy when using different training sizes and HW as leakage model. (cont.)

7

Conclusions, Limitations and Future Work

The key goal of this thesis is to improve the efficiency of deep learning side-channel attacks. To do so, we have identified problems in this research area and posed related research questions. To answer the posed questions, we have conducted experiments related to the questions which provided us with meaningful insights. In the following sections, we provide a conclusion and summary of this work, answer each research question individually, discuss the limitations of our work, and propose future work.

7.1. Conclusions

We have found that there is no need for a deep learning layer specifically designed for unprotected SCAs. From the conducted experiments, in which we compared the efficiency of the spread layer and already established layers (like MLP), we observed that the established layers are more efficient than the spread layer. Furthermore, we identified errors in the specification of the spread layer and proposed solutions to correct these errors. From the conducted experiments with the improved versions of the spread layer, we observed that the established layers are more efficient than the improved versions. Additionally, improving the attack efficiency of the established architectures even further is difficult since there is little room for improvement. We found that training network architectures using 1 000 traces is sufficient to recover the key in a few traces. Because of these findings, we conclude that there is no need for a deep learning layer specifically designed for side-channel attacks.

For the random delay countermeasure, we asked how the kernel size and depth of a network architecture influence the attack efficiency. From the conducted experiments in this work, we show that an increase in either of the two results in more efficient attacks.

More specifically, the attack efficiency is improved significantly when using a large kernel size in comparison with a small one. However, for some datasets, there is a maximum kernel size which provides the most efficient attack. Increasing the kernel size further decreases the attack efficiency. We believe that the maximum kernel size was not found for some datasets because of computational constraints. Furthermore, increasing the network's depth exhibits similar behavior as observed for the conducted experiments with the kernel size. Thus, adding convolutional layers improves the attack efficiency such that fewer attack traces are required to recover the key. However, adding too many layers is not beneficial and results in failed attacks. Additionally, we conducted experiments with noisy attack traces when there was no room for improvement of the attack efficiency. These experiments showed that increasing the kernel size and depth of the network architecture makes the networks more robust to noise. Finally, we compared the trade-off between the kernel size and depth of the network. Here, we conclude that increasing the network's depth is more beneficial than increasing the kernel size since similar results can be achieved with a deeper network in less time. However, when adding more convolutional layers decreases efficiency, increasing the kernel size improves the attack efficiency.

The experiments conducted for the random delay countermeasure have as well been conducted for the masking countermeasure. The experimental results with a masking countermeasure are significantly different from the results from a random delay countermeasure. This difference shows that CNNs are highly suitable to counter the random delay countermeasure. The largest part of the attacks on the masking countermeasure were unsuccessful. Adding additional layers to the network decreases the attack efficiency sig-

nificantly. The best results were obtained when using a single stacked convolutional layer. Furthermore, the influence of the kernel size is limited, but an increase in kernel size generally improves the attack efficiency. More importantly, we argue that MLP networks are more suitable for an attack on the masking countermeasure. In a high-order attack, an attacker combines each PoI with each other, which is similar to the mathematical functions applied in an MLP network. The difference between an attack using a CNN or MLP is significant, where the MLP outperformed CNN by several orders of magnitude.

Lastly, we have discussed a portability setting for which the traces are obtained from the same device, but the probe has been repositioned in between the measurements of the profiling and attack traces. We show that repositioning the probe has a significant influence on the attack, and makes a typical DL SCA unsuccessful. The attack fails because of a difference between the profiling and attack traces. Therefore we introduce a method that eliminates this difference. Using this method, we show an SCA is possible, and effective as well. To analyze the influence of this method for SCAs, we first develop a baseline which we use to make comparisons. Then we conduct experiments that highlight the influence of the training size and amount of epochs. The experimental results show typical behavior for SCA, providing more training traces and training for longer, both increase the attack efficiency. Thus we observe that our method allows us to perform typical DL SCAs in a portability setting.

7.1.1. Research Questions

In the previous section, we have provided a summary and conclusion of the experimental results. In this section, we answer each research question individually as posed in chapter 3.

RQ 1. *Is there a need for a sepcial designed layer for side-channel attacks on unprotected implementations?*

A 1. No, there is no necessity for a specially designed deep learning layer for side-channel attacks on unprotected implementations. We compared the attack efficiency of neural networks that utilize a spread layer and that do not. The comparison revealed that networks without the spread layer recover the key in fewer traces. Networks with the enhanced variants of the spread layer did not outperform networks without a spread layer. Additionally, the comparable architectures retrieve the key in several traces, hence improving the attack efficiency further is difficult.

RQ 2. *What is the influence of the kernel size of CNNs in a side-channel attack where the random delay countermeasure is present?*

A 2. The kernel size's influence is two-fold. Increasing the kernel size increases the attack efficiency by several orders of magnitude. However, for each data set, there is a maximum kernel size, increasing the kernel size further decreases the attack efficiency. Additionally, a large kernel is more robust to noisy traces than a small kernel.

RQ 3. *What is the influence of the stacked convolutional layers of a CNN in a side-channel attack context where the random delay countermeasure is present?*

A 3. The influence of the amount of stacked convolutional layers is twofold. Adding additional convolutional layers increase attack efficiency. However, too many convolutional layers cause the network to not converge, which causes the attack to fail.

RQ 4. *What is performance tradeoff between the kernel size and stacked convolutional layersof CNNs in a side-channel attack context where the random delay countermeasure is present?*

A 4. From our results, we conclude that adding more stacked convolutional layers achieves better attack efficiency. However, using a low kernel size like three hampers the efficiency. When increasing the kernel size big improvements in attack efficiency can be obtained.

RQ 5. *What is the influence of the kernel size of CNNs in a side-channel attack where the masking countermeasures is present?*

A 5. The kernel size's influence on the guessing entropy is minimal. Nevertheless, large kernels can make the difference between a successful and unsuccessful attack. However, a too large kernel hampers the convergence of the network such that the attacks fail.

RQ 6. *What is the influence of the stacked convolutional layers of a CNN in a side-channel attack context where the masking countermeasures is present?*

A 6. A single stacked convolutional layer provides the best results in terms of attack efficiency. Adding additional layers decreases the attack efficiency to the degree that the networks fail to recover the key.

RQ 7. *What is performance tradeoff between the kernel size and stacked convolutional layersof CNNs in a side-channel attack context where the masking countermeasure is present?*

A 7. Increasing the kernel size has the most noticeable positive influence on the attack efficiency while increasing the number of convolutional layers decreases the attack efficiency. Moreover, networks that employ additional convolutional layers need a smaller kernel size than networks with less convolutional layers. However, the kernel size has a significant influence on networks that utilize additional layers, hence the kernel size must be carefully chosen.

RQ 8. *Does the repositioning of the probe in-between measurements create a problem for a side-channel attack? And if so, what can we do to fight this problem?*

A 8. Yes, repositioning the probe in-between measurements generates a difference between the profiling and attack traces such that a deep learning side-channel attack fails. Resolving the difference of the profiling and attack traces can be achieved by normalizing each dataset individually. Conducting an attack with the normalized makes the deep learning side-channel attack successful.

7.2. Limitations

Although the work performed in this thesis has provided guidelines for future deep learning SCAs, there are some limitations. In this section we discuss these limitations.

In the deep learning domain, there is no real understanding of how a deep learning network performs classification. Hence, deep learning networks are considered as black-box algorithms. It is thus troublesome to develop network architectures that are suitable for specific datasets, and the best performing networks are usually discovered by conducting many experiments. Although in this work we distinguish datasets with comparable characteristics, there is no guarantee that the recommendations of this work are suitable for similar datasets. Our recommendations are intended to be a primary starting position that provides reasonable efficient attacks. To discover the best performing networks, it is still required to conduct experiments with different hyperparameter configurations. Moreover, we can not explain the network's predictions, hence there is no guarantee that our recommendations provide a reasonable starting point for an SCA.

Additionally, our recommendations for CNNs might not be valid for all CNN architectures since we conducted experiments using a fixed architecture skeleton. Conducting experiments that cover all imaginable architectures of CNNs is computationally infeasible, and thus suggesting to utilize a large kernel size and deep network for all CNNs' types is not possible.

As discussed in chapter 6, most literature does not perform SCAs in a realistic setting and use the same device for the profiling and attack phase. Related works state that SCAs in this setting greatly overestimate the attack efficiency. In our work, we have used datasets that were obtained from the same device, which thus suggests that we overestimate the attack efficiency of our attacks. Moreover, there is no guarantee that our recommendations are valid in this portability setting. Additionally, in a real-world scenario, where the adversary performs an attack on a real target device, there are usually some limitations. Real-world devices typically employ a countermeasure which limits the number of possible measurements. As a result, fewer measurements are accessible in the profiling phase, which toughens deep learning SCAs as well.

7.3. Future Work

In our work we have proposed recommendations that boost the efficiency of deep learning SCA, however, more work can be done to improve the efficiency and explainability of deep learning SCAs further. In this section, we discuss possible future work directions.

The deep learning community regularly introduces novel deep learning architectures that outperform state-of-the-art architectures. New research should study if these architectures outperform state-of-the-art results in the side-channel domain. Interesting architectures are those that perform well for a type of countermeasures. For example, ResNet is a DL architecture that is not studied by the SCA community. In other

domains, research has shown that this architecture performs well and could thus potentially work for DL SCAs.

From the experimental results shown in chapter 4, we observed that an MLP network, $\boldsymbol{Dense_{BN}}$, was able to successfully recover the key of $ASCAD_M$ in a few traces. We showed in chapter 5, that only a few CNNs were able to retrieve the key. These insights make us believe that MLP networks are more efficient than CNNs when attacking a masked implementation of AES. Additionally, we argued that in essence, an MLP network is capable of performing a higher-order attack. Future work could compare the efficiency of MLP networks and CNNs in diverse settings. Besides this, new studies could research a more theoretical approach to verify if an MLP network is similar to a high-order attack.

In chapter 5, we observed that applying L2-regularization is useful when networks overfit and networks trained with an L2-penalty performed the most efficient attacks. Additionally, our results suggest that L2-regularization influences the attack efficiency such that less deep CNNs are required. If future work confirms a relation exists between the used L2-penalty and network's depth, the training time could be reduced. As a result, no high-end computers are necessary to perform an attack, which would increase the attack surface. To analyze the influence of L2-regularization, similar experiments as discussed in chapter 5 can be performed.

The portability setting we have considered in this work is not the most realistic since only the probe has been moved in between the measurements of the profiling and attack traces. In the most realistic setting, the measurements are obtained from two different devices. Future work could investigate if the proposed normalization method is successful in this setting. Comparing experimental results from experiments with and without the normalization method provide insights to its success.

Furthermore, the proposed normalization method has only been used to attack an unprotected AES implementation. Since the traces of protected and unprotected implementations have distinct characteristics, the normalization method's effect is different. For example, traces obtained from an AES that implements a random delay countermeasure have higher variance, therefore a large amount of traces are required to establish a representative mean and variance. Further work should research if attacks with the normalization method are successful, and if so, determine the number of traces required to conduct efficient attacks.

Bibliography

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side—channel(s). In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36400-9.

[2] M. A. Al Faruque, S. R. Chhetri, A. Canedo, and J. Wan. Acoustic side-channel attacks on additive manufacturing systems. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–10, April 2016. doi: 10.1109/ICCPS.2016.7479068.

[3] J. Alonso and Y. Chen. Receptive field. *Scholarpedia*, 4(1):5393, 2009. doi: 10.4249/scholarpedia.5393. revision #136681.

[4] Shivam Bhasin, Anupam Chattopadhyay, Annelie Heuser, Dirmanto Jap, Stjepan Picek, and Ritu Ranjan Shrivastwa. Mind the portability: A warriors guide through realistic profiled side-channel analysis. Cryptology ePrint Archive, Report 2019/661, 2019. `https://eprint.iacr.org/2019/661`.

[5] Christopher M Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, NY, 2006. URL `http://cds.cern.ch/record/998831`. Softcover published in 2016.

[6] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-28632-5.

[7] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 45–68, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66787-4.

[8] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36400-9.

[9] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/chen`.

[10] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 156–170. Springer-Verlag, 2009. ISBN 978-3-642-04137-2. doi: 10.1007/978-3-642-04138-9_12. URL `http://dx.doi.org/10.1007/978-3-642-04138-9_12`.

[11] Debayan Das, Anupam Golder, Josef Danial, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. X-deepsca: Cross-device deep learning side channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 134:1–134:6, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6725-7. doi: 10.1145/3316781.3317934. URL `http://doi.acm.org/10.1145/3316781.3317934`.

[12] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 907–940, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR. URL `http://proceedings.mlr.press/v49/eldan16.html`.

[13] R. Gilmore, N. Hanley, and M. O'Neill. Neural network based attack on a masked implementation of aes. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111, May 2015. doi: 10.1109/HST.2015.7140247.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL http://proceedings.mlr.press/v9/glorot10a.html.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[17] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Profiled power analysis attacks using convolutional neural networks with domain knowledge. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 479–498, Cham, 2019. Springer International Publishing. ISBN 978-3-030-10970-7.

[18] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Applications of machine learning techniques in side-channel attacks: a survey. *Journal of Cryptographic Engineering*, Apr 2019. ISSN 2190-8516. doi: 10.1007/s13389-019-00212-8. URL https://doi.org/10.1007/s13389-019-00212-8.

[19] Annelie Heuser and Michael Zohner. Intelligent machine homicide. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 249–264, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29912-4.

[20] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

[21] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293, Oct 2011. ISSN 2190-8516. doi: 10.1007/s13389-011-0023-x. URL https://doi.org/10.1007/s13389-011-0023-x.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[23] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1):221–231, Jan 2013. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.59.

[24] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *CoRR*, abs/1901.06032, 2019. URL http://arxiv.org/abs/1901.06032.

[25] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):148–179, May 2019. doi: 10.13154/tches.v2019.i3.148-179. URL https://tches.iacr.org/index.php/TCHES/article/view/8292.

[26] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48405-9.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL http://doi.acm.org/10.1145/3065386.

[28] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[29] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Power analysis attack: An approach based on machine learning. *Int. J. Appl. Cryptol.*, 3(2):97–115, June 2014. ISSN 1753-0563. doi: 10.1504/IJACT.2014.062722. URL http://dx.doi.org/10.1504/IJACT.2014.062722.

[30] Liran Lerman, Stephane Fernandes Medeiros, Gianluca Bontempi, and Olivier Markowitch. A machine learning approach against a masked aes. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications*, pages 61–75, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08302-5.

[31] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 20–33, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21476-4.

[32] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard S. Zemel. Understanding the effective receptive field in deep convolutional neural networks. *CoRR*, abs/1701.04128, 2017. URL http://arxiv.org/abs/1701.04128.

[33] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 3–26, Cham, 2016. Springer International Publishing. ISBN 978-3-319-49445-6.

[34] Aravindh Mahendran and Andrea Vedaldi. Visualizing deep convolutional neural networks using natural pre-images. *CoRR*, abs/1512.02017, 2015. URL http://arxiv.org/abs/1512.02017.

[35] D. Mishkin and J. Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, November 2015.

[36] Amir Moradi, Sylvain Guilley, and Annelie Heuser. Detecting hidden leakages. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 324–342, Cham, 2014. Springer International Publishing.

[37] Information Technology Laboratory (National Institute of Standards and Technology). *Announcing the Advanced Encryption Standard (AES) [electronic resource]*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, MD, 2001.

[38] Christophe Pfeifer and Patrick Haddad. Spread: a new layer for profiled deep-learning side-channel attacks. Cryptology ePrint Archive, Report 2018/880, 2018. https://eprint.iacr.org/2018/880.

[39] S. Picek, A. Heuser, A. Jovic, S. A. Ludwig, S. Guilley, D. Jakobovic, and N. Mentens. Side-channel analysis and machine learning: A practical perspective. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4095–4102, May 2017. doi: 10.1109/IJCNN.2017.7966373.

[40] Stjepan Picek, Ioannis Petros Samiotis, Jaehun Kim, Annelie Heuser, Shivam Bhasin, and Axel Legay. On the performance of convolutional neural networks for side-channel analysis. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 157–176, Cham, 2018. Springer International Publishing. ISBN 978-3-030-05072-6.

[41] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019, Issue 1:209–237, 2019. doi: 10.13154/tches.v2019.i1.209-237. URL https://tches.iacr.org/index.php/TCHES/article/view/7339.

[42] Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, and Cecile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ascad database. Cryptology ePrint Archive, Report 2018/053, 2018. https://eprint.iacr.org/2018/053.

[43] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, June 2017. ISSN 0162-8828. doi: 10.1109/TPAMI.2016.2577031.

[44] TELECOM ParisTech SEN research group. Dpa contest (4th edition), 2013. URL http://www.dpacontest.org/v4/.

[45] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

[48] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15: 1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[49] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

[50] Biaoshuai Tao and Hongjun Wu. Improving the biclique cryptanalysis of aes. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy*, pages 39–56, Cham, 2015. Springer International Publishing. ISBN 978-3-319-19962-7.

[51] Michael Tunstall. *Smart Card Security*, pages 145–177. Springer New York, New York, NY, 2014. ISBN 978-1-4614-7915-4. doi: 10.1007/978-1-4614-7915-4_7. URL https://doi.org/10.1007/978-1-4614-7915-4_7.

[52] Huanyu Wang, Martin Brisfors, Sebastian Forsmark, and Elena Dubrova. How diversity affects deep-learning side-channel attacks. Cryptology ePrint Archive, Report 2019/664, 2019. https://eprint.iacr.org/2019/664.

[53] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *CoRR*, abs/1605.07146, 2016. URL http://arxiv.org/abs/1605.07146.

[54] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient cnn architectures in profiling attacks. Cryptology ePrint Archive, Report 2019/803, 2019. https://eprint.iacr.org/2019/803.

[55] Yingxian Zheng, Yongbin Zhou, Zhenmei Yu, Chengyu Hu, and Hailong Zhang. How to compare selections of points of interest for side-channel distinguishers in practice? In Lucas C. K. Hui, S. H. Qing, Elaine Shi, and S. M. Yiu, editors, *Information and Communications Security*, pages 200–214, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21966-0.

[56] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Report 2005/388, 2005. https://eprint.iacr.org/2005/388.

A

CNN Results

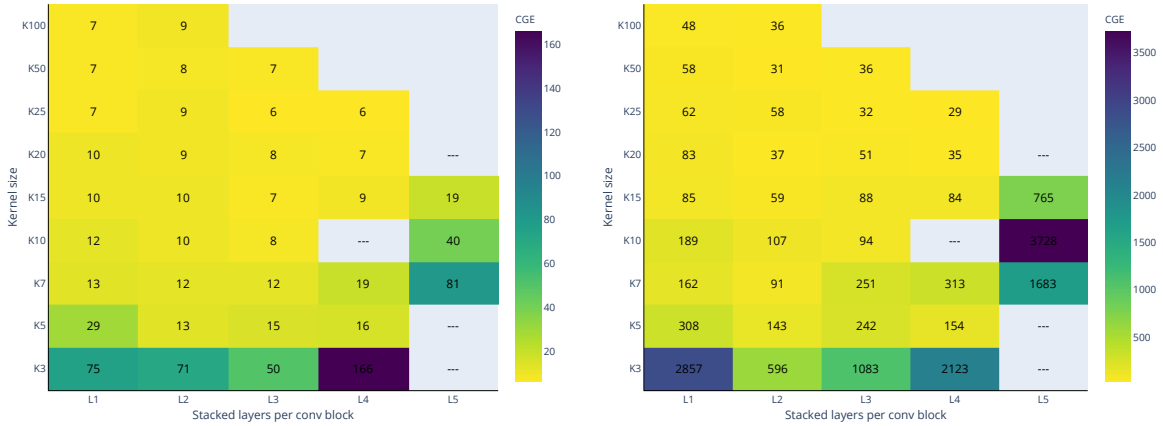
Here we show the experimental results for the architecture $\boldsymbol{VGG}_{l,k}$ with an L2-penalty $\lambda = 0.05$ using the noisy attack traces, and $\boldsymbol{VGGMax}_{l,k}$ using the noisy attack traces.

A.1. L2-regularization

In Figure A.1 and Figure A.2, we show the *CGE* and accuracy of $\boldsymbol{VGG}_{l,k}$. From the *CGE* we see for each depth of a network that, increasing the kernel size provides lower *CGE*, and thus we recover the key faster. When adding more noise to the attack traces we observe that the shallow networks with a big kernel size perform the best. This is also visible in the plots of the accuracy because the accuracy is highest with the larger kernel sizes.

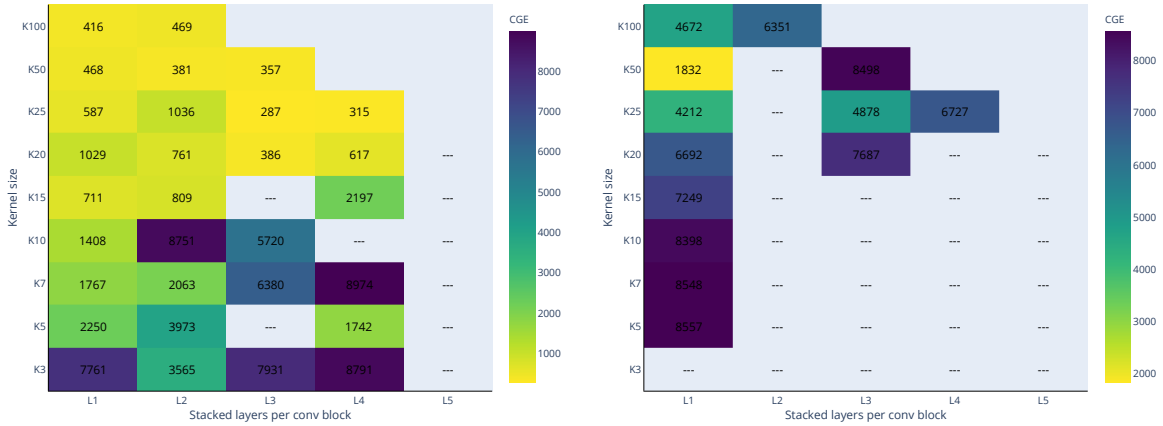
A.2. Max Pool

In Figure A.3 and Figure A.4, we depict the *CGE* and accuracy of the networks $\boldsymbol{VGGMax}_{l,k}$ with noisy attack traces. Here we observe that as we increase the noisy the networks with a larger kernel size perform the best. However, here we observe that the deeper networks also perform better. We believe these results occur because we did not apply L2-regularization in this setting.



(a) $\alpha = 0.25$

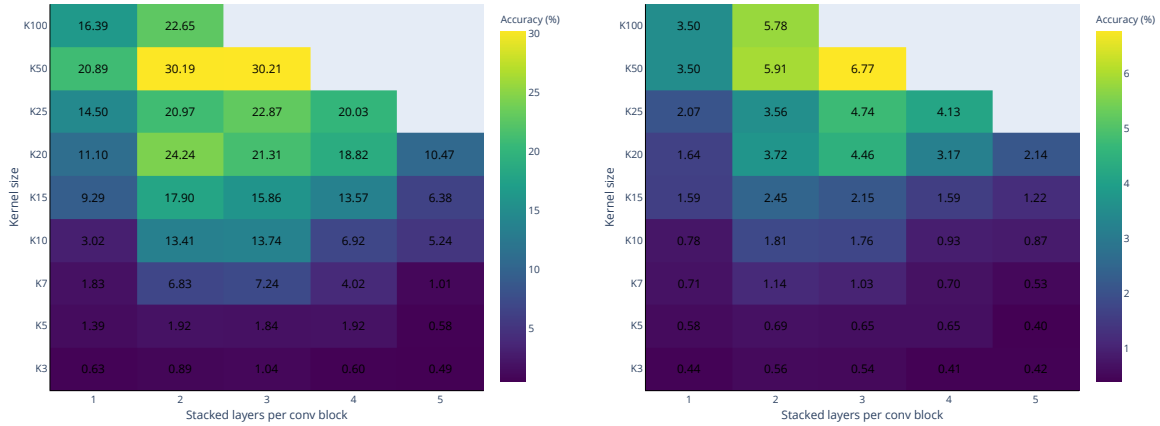
(b) $\alpha = 0.5$



(c) $\alpha = 0.75$

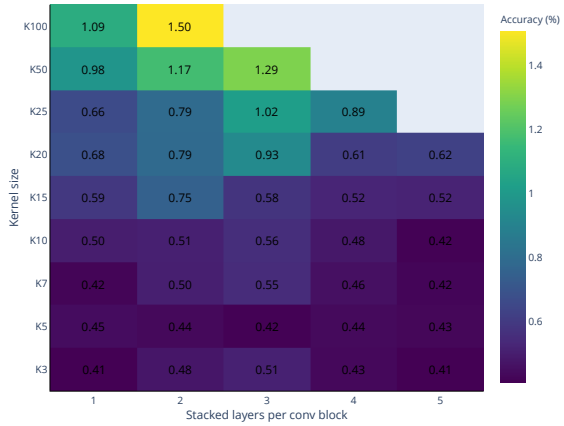
(d) $\alpha = 1.0$

Figure A.1: *CGE* of the $\boldsymbol{VGG}_{l,k}$ trained with a L2-penalty of $\lambda = 0.05$ on *RD* and using noisy attack traces (noisiness defined by α)

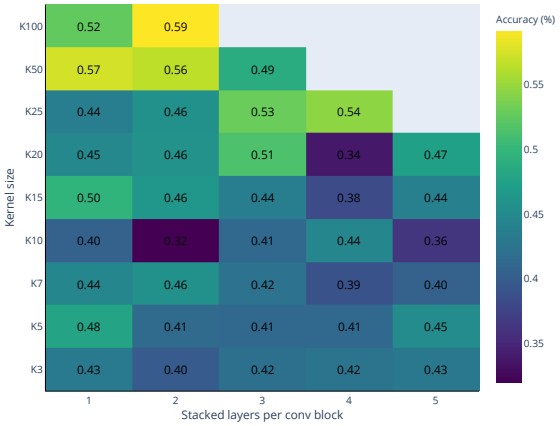


(a) $\alpha = 0.25$

(b) $\alpha = 0.5$

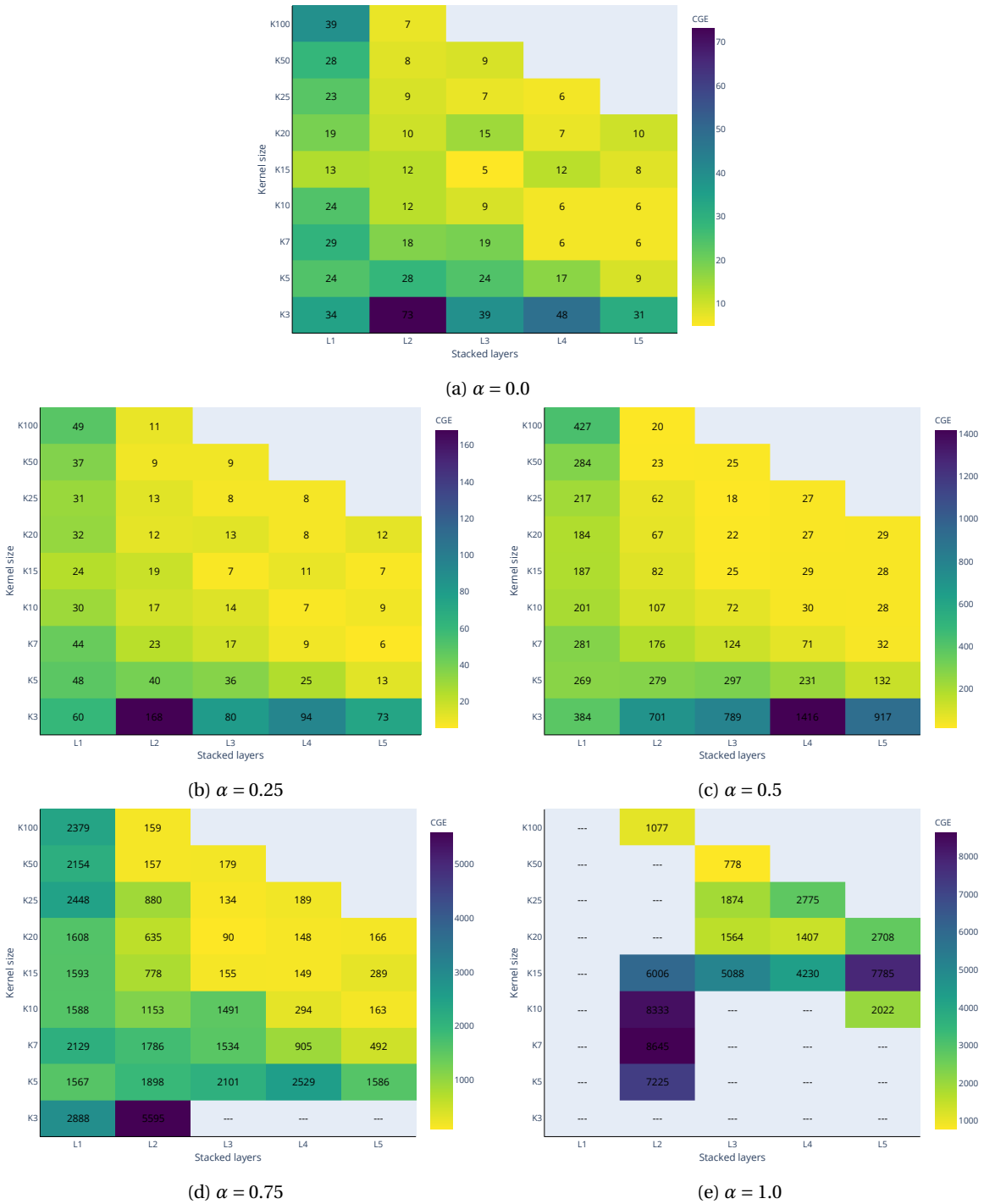


(c) $\alpha = 0.75$



(d) $\alpha = 1.0$

Figure A.2: Accuracy of the $\boldsymbol{VGG}_{l,k}$ trained with a L2-penalty of $\lambda = 0.05$ on *RD* and using noisy attack traces (noisiness defined by α)



(a) $\alpha = 0.0$

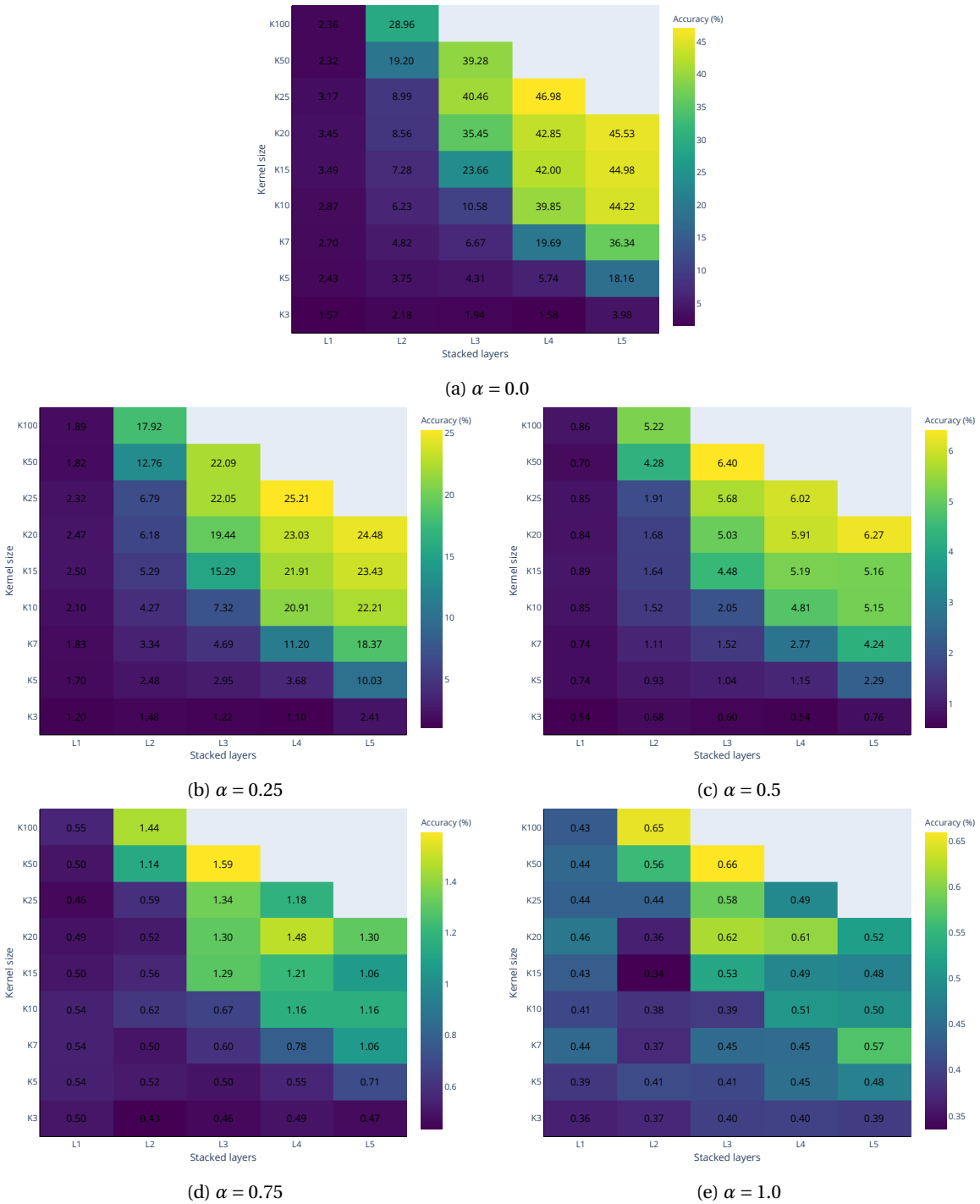
(b) $\alpha = 0.25$

(c) $\alpha = 0.5$

(d) $\alpha = 0.75$

(e) $\alpha = 1.0$

Figure A.3: *CGE* of $\boldsymbol{VGGMax}_{l,k}$ on *RD* using different noise factors α for the attack traces



(a) $\alpha = 0.0$

(b) $\alpha = 0.25$

(c) $\alpha = 0.5$

(d) $\alpha = 0.75$

(e) $\alpha = 1.0$

Figure A.4: Accuracy of $\boldsymbol{VGGMax}_{l,k}$ on *RD* using different noise factors α for the attack traces