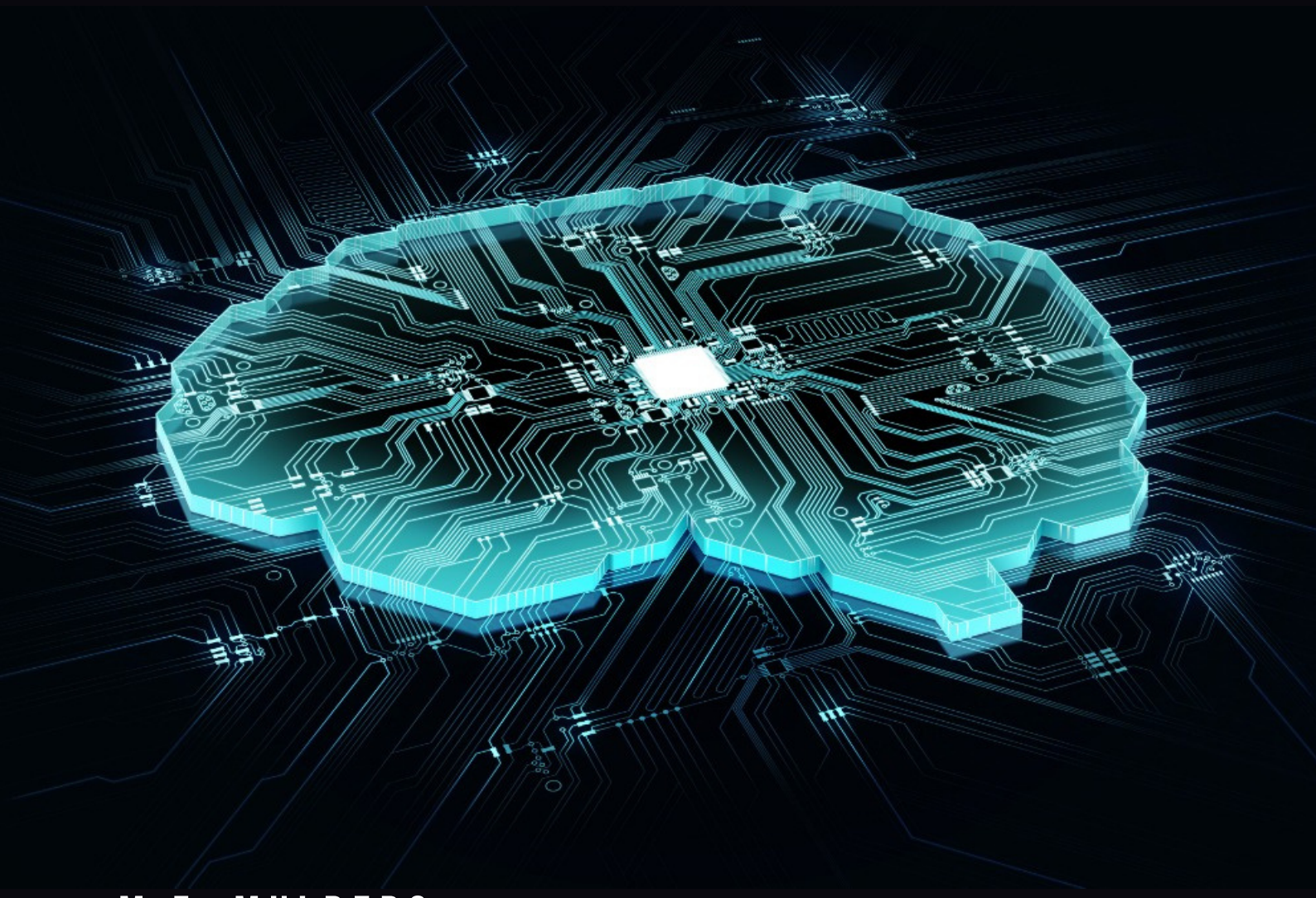
ON THE SEQUENTIAL DATA MODELS IN SIDE-CHANNEL ANALYSIS

RNN, LSTM, GRU Hyperparameters,
Autoencoder and Embedding Layer



M.F. MULDERS

TU Delft

On the Sequential Data Models in Side-Channel Analysis

RNN, LSTM, GRU Hyperparameters, Autoencoder and Embedding Layer

by

M. F. Mulders

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on Thursday November 12, 2020 at 10:00 AM.

Thesis committee:	Dr. S. Picek	TU Delft, Supervisor
	Prof. dr. ir. R. L. Lagendijk,	TU Delft, Chair
	Dr. P. K. Murukannaiah,	TU Delft

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

TU Delft

Abstract

A side-channel attack is performed by analyzing unwanted physical leakage to achieve a more effective attack on the cryptographic key. An attacker performs a profiled attack when he has a physical and identical copy of the target device, meaning the attacker is in full control of the target device. Therefore, these profiled attacks are known as the most powerful attacks in the side-channel analysis. This physical leakage is analyzed by machine learning and, in the last years, mostly deep learning, which both are used as a profiling tool to perform a side-channel attack. The best known deep learning technique for side-channel analysis at this moment is the convolutional neural network (CNN).

However, this thesis investigates a well-known deep learning model that is never used before in side-channel analysis. The deep learning models RNN, LSTM, and GRU are tested and evaluated to look for the best hyperparameters. We show the influence of different models, amount of layers, dropout, activation function, units, recurrent dropout, and batch sizes in the experiments. We also show that using different sequence length gives a speedup in training. To reduce the sequence length, we use a linear regression technique. After that, we show that sequential data models are a suitable alternative for side-channel analysis; however, their results do not surpass the CNNs.

After this, we experiment with an autoencoder as a preprocessing algorithm to ”clean” noisy traces. We show that the LSTM autoencoder easily removes a hiding countermeasure with noise. However, a hiding countermeasure with delay is more challenging for the LSTM autoencoder. Combining both countermeasures seems impossible for the LSTM autoencoder. The performance we see when cleaning the traces also affects the guessing entropy.

Lastly, we use an embedding layer as the first layer for MLP, CNN, and a sequential data model in the side-channel analysis. We experiment with different output dimensions and conclude that an embedding layer is a valid alternative to change the data dimension when using an MLP or a sequential data model.

Preface

In front of you lies my hard work that should finish my master at the TU Delft. Starting with this thesis around October 2019 makes it almost a year of hard work. When I started, I expected to finish before the summer holidays. However, working from home and finding a new working routine took more time than expected. Nonetheless is reaching the finish line of a master thesis an achievement that I should be proud of. I would not have finished this research project without some people's help, which I like to thank here.

First of all, I would like to thank my daily supervisor, Stjepan Picek. Having you as my supervisor was an interesting choice. Meeting with you almost every week and discussing both research work as personal topics. Bragging that you are good at asking "smart" questions, after which I had enough to do for the rest of the week. For me, the combination of discussing research work and making fun of everything else kept me motivated to keep on working. So a big thank you for helping me to put down this achievement.

Secondly, I want to thank everyone at the university who has helped me when working on my thesis. First of all, Marina, for our short Friday meetings, preparing me for the meeting with Stjepan. But also for proofreading the whole story and correcting most of the grammar issues. Moreover, the other master students, Tim, Clinton, Jorai, Cas, Daniel, Felix, Jehan, Roy, and Tristan, with whom we held the competition being first to have a place on the 6th floor and had terrific lunches/coffee brakes. And of course, the most critical person, Sandra, who was able to give our coffee when we needed it.

Finally, I want to thank the people who have actively supported me during my studies in Delft. These are mostly my parents, who even made it possible for me to study and gave me the space to find the right study. But also my brothers, friends, and girlfriend. Who kept asking about how I was doing with my research project but had no clue what I was doing.

Without the people above, I could not have finished this thesis. Even more, to graduate at the university where I have studied for eight beautiful years.

M. F. Mulders
Delft, November 2020

Contents

1 Introduction

> "Something that tells you something about something without knowing that something."

It was this sentence on the slides of a lecture during System Security that got our attention. The word *something* has been used many times, which could be considered annoying. However, in this instance, it means something powerful; an attacker can retrieve much valuable information and perform an effective attack, a side-channel attack. This thesis mainly dives into the practical side-channel attack. Even though the quote sounds very abstract, the usage of a side-channel attack is well known. For example, an old school movie, where a thief tries to break into a safe. The thief uses a stethoscope to listen for the pin to fall into the right position. The sound produced by the lock is an undesired side-channel that can be compromised and helps the thief find the correct key even faster.
To get a hold of what a side-channel attack is, we could look at it even in a broader context. What about a good friend who has a secret for us? Maybe he is in love with someone we know, but he is not willing to tell us. There are many social side-channels that we can use to retrieve information from him. For example, him blinking his eyes or him brushing through his hair. This will give the attacker more knowledge of his secret.

The above example shows how an attacker retrieves valuable information with common sense and general knowledge. Since the number of systems connected to the internet increased exponentially, the domain for side-channel attacks also got more popular. The first academic side-channel attacks are based on statistical methods and date back to the late 1990s [30]. Nowadays, examples are above our imagination. With the use of a computer, more specifically, with machine learning and deep learning, we see a new way of side-channels that have not been seen before. Machine learning in computer science has made it possible to see patterns and correlations in big data without human interaction. Therefore, it is possible to collect big datasets and use machine learning techniques generated by a computer to find the dataset's relevant information. A contrived action involving constant pursuit game started between producers of systems and adversaries looking for relevant information that those systems were leaking. The defenders started to use countermeasures like hiding and masking; the attackers use deep learning techniques to counter those countermeasures. This ongoing pursuit is still taking place, and with the computer hardware getting cheaper every year, even the amount of time of an attack is not an issue anymore.

A side-channel attack can be divided into two types: profiled side-channel attacks and non-profiled side-channel attacks. Where the non-profiled attack makes simple assumptions, the profiled side-channel attack is more powerful. In the profiled setting, the attacker has a "copy" of the target device. We can generate this clone because we know the possible hardware is used (version and firmware) into the under attack systems, neglecting the environment noise when capturing the traces. The strength of this side-channel attack is based on the correctness of the clone device. This attacker uses this physical copy of the device to train a profiling model. Then this trained model could be used to attack the real device. In this setting, the attacker has full access to the copy device. In the non-profiled case, the attacker has no access to the victim's training device and has less information.

As described above, an attack is possible, and attackers are at the winning hand. However, from the scientific domain, we are always at the defending side of the story. The only possible way to defend

ourselves against attacks is to know, or at least get a glimpse, what an attacker is capable of. To get that knowledge, we perform many attacks and use various models. Then we can try to introduce countermeasures. These countermeasures are used to make it harder for the attacker to profile the traces. When we find out how good a specific countermeasure is performing against a single attack, we can determine how good the designed countermeasure is. We do this by creating a baseline and find out if another countermeasure is performing better. This baseline that is needed for defending is what the scientific community is publishing. In current work, we see considerable interest in using deep learning techniques, specifically convolutional neural networks. These techniques are powerful and extremely good at dealing with countermeasures such as masking and hiding [57]. Therefore, there is now a shortsightedness approach to convolutional neural networks. Everyone is focusing on these models, and with that, most people forget to take a look at the big picture of the deep learning domain. One model that is neglected is the recurrent neural network. Moreover, everyone at side-channel related conferences said they want to do further research in recurrent neural networks. Meaning, researchers are addressing the topic a lot, but only one research paper has been published.

That is where this thesis finds its place. This thesis looks at a completely different deep learning technique that rarely has been used before in the side-channel domain. Moreover, the thesis will be the first deep dive into these new models. The models used in this thesis are part of the sequential data models; these are recurrent neural networks, long short-term memory, and gated recurrent unit. These models are mostly used in the natural language processing domain [11, 73, 75]. There is another domain where these models are used, namely the health care domain and especially on electroencephalogram (EEG) data [16, 33, 47, 50, 51, 64, 68]. From the perspective of the side-channel domain, using these sequential data models is entirely new. Only one paper [38] used the long short-term memory model. Their results are only compared to the baseline and were relatively poor. However, a good explanation was not given. It was just a run and evaluate, with no further research on the hyperparameters or explaining why the models performed poorly.

The side-channel domain is a small group of researchers exploring a specific topic. Meaning, there are not many datasets to work on, and most information to make better attacks is publicly known. This results in the fact that most publications compete against each other like S. Picek said: ”Squeezing out the last molecule of improvements.” People are competing with each other by reducing the size of a model or time of training. For the lasts years, there has not been a big jump of research topics inside this domain.

Finally, to the best of our knowledge, *sequential data models* is not a definition used in literature. In literature, the collective name for recurrent neural networks, long short-term memory, and the gated recurrent unit is RNN. However, to make a clear distinction between the use of the recurrent neural network and the three in total, we use in this thesis the term sequential data model as a collective name for the recurrent neural network, long short-term memory, and gated recurrent unit.

1.1. Research Question

The goal of this thesis is to take a broader look into the sequential data models. We look into three sequential data models: the recurrent neural network, the long short-term memory, and the gated recurrent unit. How do these models perform on side-channel data? What is the best way to use them? Moreover, what should be done to let them perform at their best? There is much knowledge for these models on different data types, but not for this specific data used with side-channel attacks. Therefore, general research questions in this thesis will be:

- How can sequential data models be used in the side-channel analysis?
- What kind of preprocessing techniques from the sequential domain can be used to improve the quality of the side-channel attack?
- What kind of natural language processing techniques can be used to make a side-channel attack more efficient?

More specific research questions and why these questions are novel and needed for side-channel analysis are stated in the related work (Chapter 3).

1.2. Scientific Contribution

This thesis answers the questions mentioned above. This is done by carefully looking into the research of other domains and finding the strengths that could also work for the side-channel domain. Furthermore, we evaluate different setups for the sequential data models and compare these with the current baseline. We give a final recommendation of applying these models in side-channel attacks. In short, the scientific contributions of this thesis are:

- A well-substantiated advice in using or not using sequential data models in side-channel attacks, where the advice is also explained.
- A new baseline for using sequential data models in side-channel attacks in datasets with masking and hiding countermeasures.
- A new way of preprocessing datasets for side-channel attacks where the sequence length is reduced, but the information is still present.
- A novel way of removing noise from datasets in the preprocessing stage of the dataset.

1.3. Outline

This thesis is a broad study of the above-stated research question. The rest of the thesis is outlined as follows: Chapter 2 covers the background of this thesis. We explain the side-channel analysis, AES, machine learning, deep learning, recurrent neural networks, long short-term memory, gated recurrent unit, and the datasets used. As stated in Chapter 3, all related work is discussed. It will include information about state-of-the-art side-channel attacks, why it is relevant to look into another model, and why deep learning is used in side-channel attacks. In the following chapters, we discuss the contribution of this thesis. In Chapter 4, we evaluate the use of sequential data models and gather evidence for our advice. In Chapter 5, we look into using autoencoders with long short-term memory to clean the data and remove the noise. In Chapter 6, we look into using embedding as a preprocessing technique to improve the sequential data models, which we also tested with state-of-the-art multilayer perceptron and convolutional neural network. In Chapter 7, we answer the research questions from section 1.1; there, we also address future work for this thesis. In the attachment is a section with the implementation details and example code for reproducibility.

2

Background

We start with an introduction of advanced encryption standard(AES), after which we also explain the known attacks on AES. We consider only the AES algorithm in this thesis because we see most side-channel attacks attacking this encryption standard. From there, we take an in-depth look into the side-channel analysis, explain types of side-channel attacks, and the countermeasures that are known in the field. The third section covers an explanation of machine learning, how to evaluate, and what kind of hyperparameters we can tweak when dealing with machine learning. Also, the multilayer perceptron and the neural networks are explained. The fourth section explains the deep learning models used in this thesis. These are recurrent neural networks, long short-term memory, gated recurrent units, convolutional neural networks, and an autoencoder. The fifth section examines specific information about natural language processing techniques. This thesis uses many techniques used in the natural language processing domain and, therefore, particularly from recurrent models. The last section explains the different datasets that are used in this thesis.

2.1. Cryptography - Advanced Encryption Standard

The Advanced Encryption Standard, developed by Joan Daemen and Vincenet Rijmen in 2001 and a subset of the Rijndael block cipher [14], is the new common encryption standard used in the industry after winning a competition of the US National Institute of Standard and Technology. The Data Encryption Standard (DES), published in 1977, has been superseded by the Advanced Encryption Standard (AES). AES algorithm is a symmetric-key algorithm, meaning it uses the same key for encryption and decryption. The Rijndael algorithm supports key sizes of 128, 160, 192, 224, and 256 bits and block sizes of 128, 160, 192, 224, and 256 bits. However, AES's specific implementation is used with a block size of only 128 bits and three different key sizes, namely 128, 192, and 256. The key size of the AES decides how many rounds are used. AES can have 10, 12, or 14 rounds, respectively, for the 128, 192, and 256-bit keys.

AES encryption consists of four main parts.

1. *KeyExpansion*

   • Round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.

2. *Initial round*

   • Add Round key: A bitwise XOR is used to add each byte of a state with a block of the round key

3. *Rounds* (9,11 or 13 rounds)
   Each round consists of four steps.

   • SubBytes: a non-linear step where each byte is replaced by another byte using an 8-bit $Sbox$.

- Shiftrows: the last three rows of the internal state is shifted a certain number cyclically.
- MixColums: A linear step where matrix multiplication is used as Galois Field 2^8.
- AddRoundKey: A bitwise XOR is used to add each byte of a state with a block of the round key

4. *Final round* (adding the total round up to 10,12 or 14).
   - SubBytes: a non-linear step where each byte is replaced by another byte using an 8-bit $Sbox$.
   - Shiftrows: the last three rows of the internal state is shifted a certain amount of times cyclically.
   - AddRoundKey: A bitwise XOR is used to add each byte of a state with a round key block.

A visual example of the AES encryption and decryption scheme can be found in Figure 2.1.

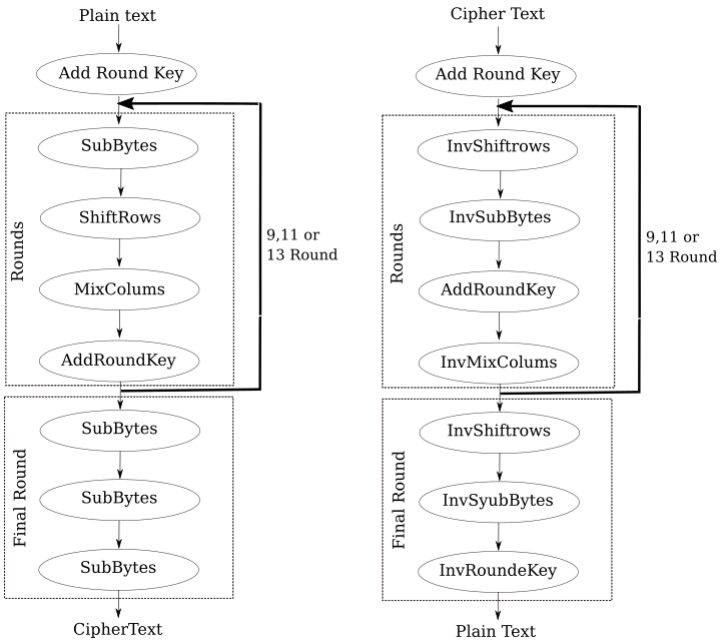


Figure 2.1: AES Encryption (left side) and Decryption (right side).

2.1.1. Existing Attacks on AES

Breaking a cipher in a cryptographic context means anything faster than performing a brute-force attack. For that reason, a cipher can be called broken before an attacker assumes the cipher to be broken, i.e., it still takes too much time to break the cipher. A brute-force attack is an attack where the attacker submits all the possible passwords or keys to retrieve the correct one. The attacker has to calculate every possible combination that could make up the secret information. As the length of the password increases linear, the time of the attack increases exponentially. In the past, some attacks have been proven to be faster than a brute force attack. The first key-recovery attack on AES was launched in 2011 [7]. This attack was a biclique attack and resulted in being four times faster than a brute force attack. For example, the key recovery attack on AES-128 can be computed with a computational complexity of $2^{126.1}$. However, all the known attacks are not computationally feasible.
Side-channel attacks that attack the algorithm do not consider the cipher as a black-box. Instead of that, they attack the cipher on the hardware or other software systems that leak data. An example of a published attack was in December 2009, where differential fault analysis was used. That attack allows key recovery of the key with a computational complexity of 2^{32}.

2.2. Side-Channel Attacks

A Side-Channel Attack (SCA) is an attack based on information gained from the implementation of the hardware. The attack is thus different from most attacks because it does not use the implemented algorithm's weakness. For a side-channel attack, the attacker needs some knowledge of the system. In case the adversary has the profiling device, the attack is called a profiling side-channel attack (explained in 2.2.1). In case the adversary does not possess a clone device, the attack is called a non-profiling side-channel attack (described in 2.2.2). In the last case, the attacker only has access to the physical leakages of the device. Within every section, the different attacks are explained. An overview of which attack belongs to which category can be seen in Figure 2.2.

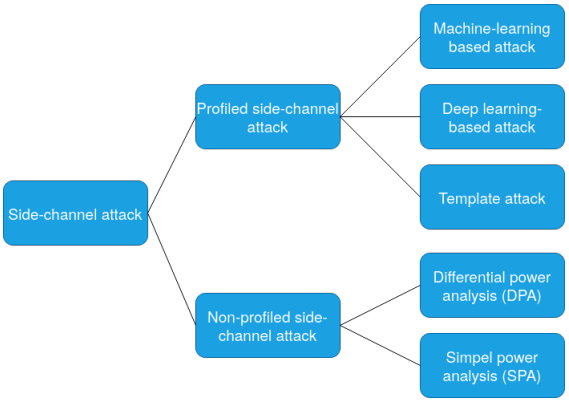


Figure 2.2: Categories of side-channel attacks.

To implement a side-channel attack, there needs to be some kind of leakage. These different forms of physical leakages a device is generating will be used to attack. The following list includes some examples of different types of leakages a device has and how this leakage is measured:

- Timing leakage:
  This attack is based on how much time it takes for the device to compute various computations. The physical leakage is thus the timing of a computation.
- Power Consumption leakage:
  This attack is made by measuring the power consumption of the hardware during the computation of the cipher. The physical leakage is thus power consumption.
- Electromagnetic leakage:
  This attack is based on electromagnetic radiation. Because of the electromagnetic radiation the device is leaking, an attacker can directly get plain texts and other information. The physical leakage is thus electromagnetic radiation.
- Acoustic leakage:
  This attack looks like a power consumption attack but uses the sound produced instead of the power. The physical leakage is thus sound of computation.

2.2.1. Profiling SCA

With this analysis, the adversary owns a profiling device. The attack is known as the most powerful SCA in the literature and consists of two steps. The adversary owns a copy of the target device (profiling device). The target device is then profiled and characterized by manipulating the data and capturing the leakages and the clone device's behavior. In the second step, the adversary performs an attack with the target device's model (profiling model) in the key-recovery phase. The set of possible attacks are then: template attack[10], linear regression analyses [6], and machine learning-based attacks[23].

2.2.2. Non-profiling SCA

Non-profiling side-channel analysis, compared to profiling SCA, is much weaker. The adversary has access to the target device's physical leakage and, thus, not the actual device itself. Therefore the adversary tries to recover the secret key by performing statistical calculations or relate the countermeasures

that can be roughly divided into two categories to detect dependency between leakage measurements and the secret key. The set of possible attacks for this method are then: Differential Power Analysis (DPA)[31], Correlation Power Analysis (CPA)[8], and Mutual Information Analysis [15].

2.2.3. Countermeasures

The state-of-the-art attacks result in the fact that several countermeasures are being placed in hardware by the industry. These countermeasures are set in place to make it harder to execute an effective and efficient side-channel attack. We divide the type of countermeasures into two groups. First, a physical countermeasure, making it harder to obtain traces from a device. The other group contains software and hardware related countermeasures which eliminate the relation or correlation between the leaked information and the intermediate value.
The first group in which physical countermeasures are practiced is called *shielding*. In this group, the countermeasures make it harder to collect the traces from the device. With this type of countermeasure, an attacker needs more knowledge about the device. If the attacker owns more advanced and more expensive equipment to capture the traces, he will still capture the traces. Therefore, this countermeasure has a limited effect but will most likely work against script kiddies. The second group, which is more implementation based has two types of countermeasures:

- The first countermeasure in this group is called *hiding countermeasures* [41]. A hiding countermeasure is a countermeasure that eliminates or reduces the information leaked by the system. There we want to hide the real traces inside the hardware model. So the goal of a hiding countermeasure is to change characteristics directly. An example of doing this is to add random noise to the signal or let every computation cost the same amount of power. More common techniques also included random delay or shuffling. Random delay countermeasure is a technique where timing difference is created between the data points. Random operations fill in the gaps that are created by this method without any meaning for the encryption. These hiding countermeasure aims at decreasing the Signal-to-Noise ratio (SNR) because the method increases the noise and decreases the signal. However, some patterns that identify the trace can still be present in the captured data. For example, convolutional neural networks can deal with these hiding techniques and, therefore, may find the pattern that is in the trace.

- The second countermeasure in this group is called *masking countermeasures* [42]. This countermeasure can eliminate the sensitive leakages from the model. A common way to mask an AES encryption is to alter the output of the $Sbox$ with boolean values [61]. Normally we XOR the input with the round key and use that with the $Sbox$; now, a mask is added after the $Sbox$ operation.

$$\begin{aligned} \text{Normal situation: } Out &= Sbox[in_j \oplus rk_j], \\ \text{Masked situation: } Out &= Sbox[in_j \oplus rk_j] \oplus mask_j \end{aligned} \tag{2.1}$$

  In Equation (2.1) is in_j the input at round j, and rk_j the round key at round j. Masking can be done in higher orders. Then we speak about a higher order mask with the order of n. In these higher orders masks, n random masks are picked to mask the data like in the masked situation in Equation (2.1), masked situation. A higher-order attack can still break a masked implementation. This attack should be of the order $n + 1$. The attacker could then select $n + 1$ points of interest and try to correlate these points with each other.

To conclude, many countermeasures are designed and integrated, but the reality is that every countermeasure can be dealt with by the attacker. In the current state of side-channel analysis, it is advised to combine the above-discussed countermeasures. The complexity of the attack increases when using different countermeasures. This is visible in Figure 2.3.

2.2.4. Guessing Entropy

In every evaluation problem, there is a need for a metric. Usually, machine learning problems use metrics such as accuracy, precision, or recall. However, we see that those metrics do not evaluate a side-channel attack well enough. Therefore we use a metric called Guessing Entropy (GE) [62].
The GE of a model indicates the average number of key guesses that need to be tested before the

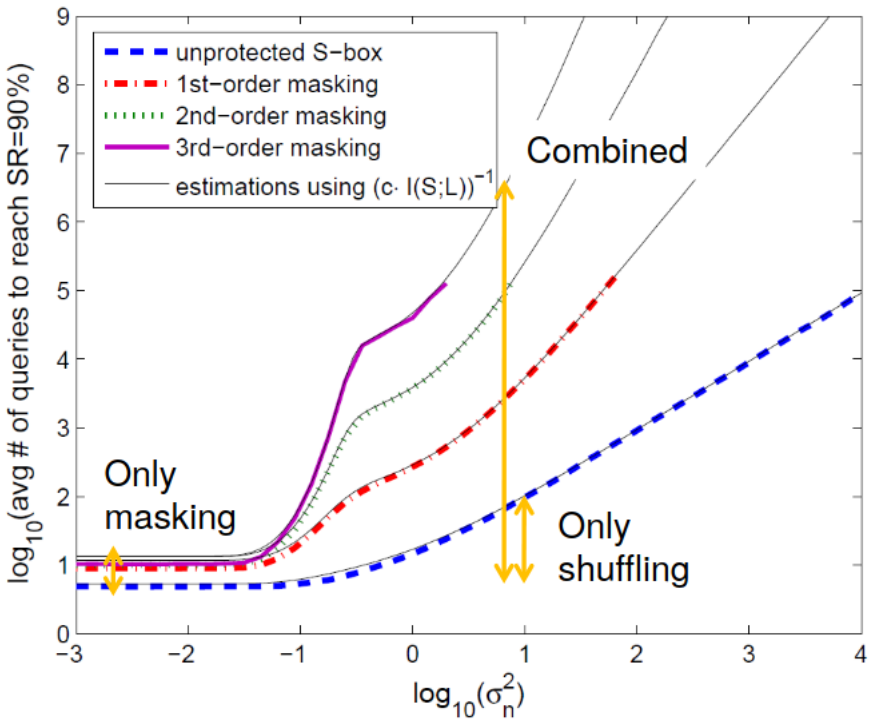


Figure 2.3: Visualization of how different techniques increase the complexity of the model. On the X-axis is the number of traces related to the countermeasure. On the Y-axis is the average number of queries to reach a SR of 90%. The figure was found at [43].

attacker recovers the correct key byte. When attacking the trained model, the attacker creates an array (with the size of the keyspace) with the key guesses sorted descending on all the probabilities for each key. This means the key at position 0 has the highest probability to be the right key. To calculate this GE over multiple traces, the log function of the probability is calculated. In this setting, the amount of traces is optimized by getting the correct intermediate value at position 0 in the array. We write down the amount of traced needed before reaching that setting. Meaning a GE of 0 after using 200 traces is better than reaching a GE of 0 after using 400 traces because the model needed fewer traces to get the correct key at position 0. A GE that never reaches 0 means that the model is not able to find the right key.
However, what should be noted is that GE could be dependent on which traces it sees first. Some traces could be more helpful for finding the right key than others. It would make a difference whether these traces are in the beginning or at the end of the attack. To overcome this problem, we always calculate the GE with a different permutation of the attack traces. This is done 100 times with random permutations. Every single GE that is calculated is called the Partial Guessing Entropy (PGE). Then the average GE of the 100 PGEs is calculated to come with a final GE for a particular experiment.

2.2.5. Leakage Models

SCA does have a variety of leakage models. In the previous section, the conclusion was made that the device is leaking information. The difference with a leakage model is how we notate the leakage, how the leakage is captured stays the same. This thesis uses two different leakage models, which are commonly used in literature, which are hamming weight and intermediate values.

Hamming weight (HW) is a way of notating values and named after Richard Hamming [17]. The hamming weight is calculated by counting the number of ones in a binary representation of the leaking value. The advantage of the hamming weight leakage model is that it has fewer classes in the classification phase, which results in a smaller training complexity. Because of the fewer classes, this dataset also suffers from class imbalance. Equation (2.2) is an example of calculating decimal values to hamming weight. In this example, we can see a class imbalance since most of the values have a

hamming weight of 4.

$$\begin{aligned} HW(131) &= HW(10000011_2) = 3, \\ HW(106) &= HW(01101010_2) = 4, \\ HW(45) &= HW(00101101_2) = 4, \\ HW(230) &= HW(11100110_2) = 5, \\ HW(32) &= HW(00100000_2) = 1. \end{aligned} \tag{2.2}$$

Intermediate value (ID) is the most standard way of notating the leakage model and most commonly used in software models. In this case, the model has 256 classes, which are equal to the intermediate key values. Here we classify directly to the key values and do not suffer from class imbalance. However, because of the enormous class vector space, the training complexity increases and makes it harder for the model to classify the right traces to the right ID value.

2.3. Machine Learning

Machine learning (ML) is a mechanism that has been used a lot in the past twenty years. Humans are the ones that program machines and make algorithms to follow a certain set of ordered instructions. If humans instruct these machines to learn from the past data, the machines can learn and adapt to these past experiences way faster. This phenomenon is called machine learning. A machine that is solving a task without having specific instructions on how to solve the task, but learning itself to give instructions based on past data. In computer science, we divide these machine learning algorithms roughly into three groups. The first group is *supervised learning*, the second group is *unsupervised learning*, and the last group is *reinforcement learning*.

- In supervised learning, a model learns from the data using features in the data to identify the difference between different objects. In the training phase, the model has access to the different labels of the objects. The model does learn what features correspond with which label. Then, when the model sees a new object, it looks into that object's features. It will classify the new object with a label from the training phase. In supervised learning, new unseen data has always to be classified to the known classes.
- In unsupervised learning, a model also learns from the data using features present in the data. The big difference between unsupervised and supervised learning is that we do not have labels for the input data. The machine will cluster the different objects according to their features and tell afterward which objects look the same. Here the machine can make extra 'labels' which can create more classes but also increase sparsity.
- In reinforcement learning, the model works on the principle of feedback. When we give an object to the model, it will identify the object based on features and characteristics. The model puts a label on the object, which is followed by giving feedback to the model. If the object was not labeled correctly, the machine will adapt its identifiers to the right values and, therefore, will better recognize comparable objects.

Thus, a machine learning model is something that maps an input to an output. Where every object in that input has a feature vector of n features which is represented as $\vec{x} = (x_1, x_2, ..., x_n)$. In the side-channel domain, this object will be a trace holding at every time step a feature with side-channel information. This feature depends on which leakages are measured. The label of these traces is the intermediate values or intermediate keys. Those labels are typically represented as y. A machine learning algorithm tries to create a function that maps those inputs to the outputs. This function is represented as follow: $f : X \rightarrow y$. There are different machine learning techniques. In this thesis, we will mainly focus on the neural networks and the classification problem.

We can divide machine learning classification problems into four different categories. See Figure 2.4 where the input is the black box. The steps in the algorithm are the blue box, and the yellow box is the output. In the following itemize, we explain the four different categories of the figure.

- One to one classification, where the input consists of one value, and also the output is singular.

- One to many classifications, where the input consists of one value, and the output can be classified into more classes.
- Many to one classification, this is where the input consists of many inputs, and the output is a singular value. For example, this is a problem with a single trace as input (where the trace has multiple timesteps), and this has to be classified into one intermediate key value.
- Many to many classifications, this is where the input and the output consist of many values. We see this type with translation problems where one sentence with multiple words is translated to the other sentence with multiple words.

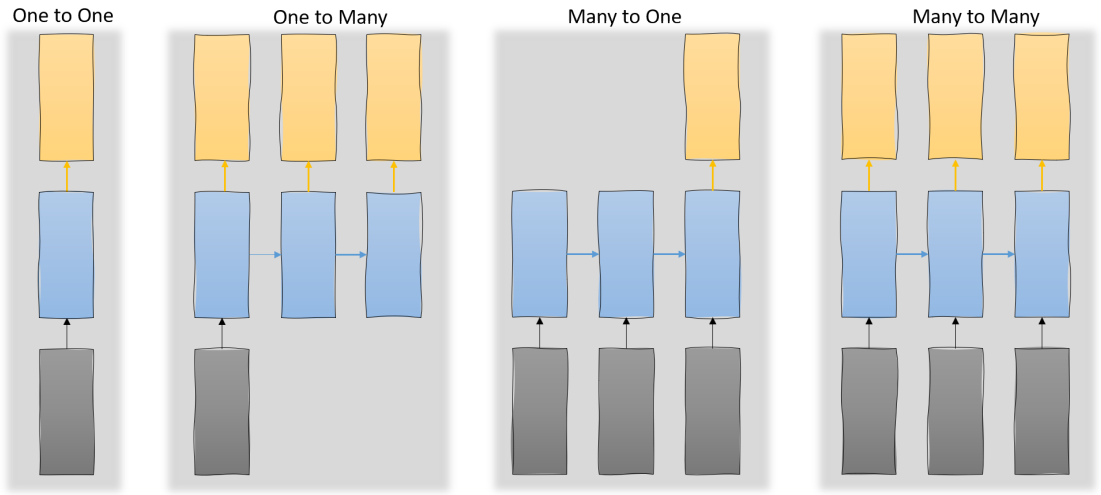


Figure 2.4: Graphical visualization of different categories of classification problems.

For every training, the model can be tweaked. These configurations that are given to the model before training are called *hyperparameters*. So, the hyperparameters are set at the beginning of the training phase and can be optimized by the user. This thesis is a broad study of what are the right hyperparameters for sequential data models. Possible hyperparameters are dropout, recurrent dropout, amount of layers (more broadly architecture), batch size, epochs, and activation functions. Increasing the number of epochs will increase the number of training runs. Therefore we could find the optimal value between overfitting and underfitting.

2.3.1. Neurons

Since we focus on artificial neural networks in this thesis, we explain neurons. In this subsection, we shortly discuss these neurons. The idea of these neurons is taken from the brain's function, and they are the building units of the artificial neural networks. These neurons take specific inputs, calculate the weighted sum and then apply an activation function, generating an output used for the next neuron/layer. These neurons' inputs can be the output of a previous layer or the original input from the dataset. The weights are then applied and will be updated every batch size according to the optimization algorithm or in every epoch of the training phase. The neuron then uses an activation function, and, based on that, it fires its outputs to the next layer. The bias value, which can be seen in Figure 2.5, is added to the neuron before calculating the activation function. This bias term is also something that is learned but is a constant factor that is added to the neuron. When an input of the dataset is all 0, the neuron's output is always zero. However, when the output should be 1, this can be compensated with the bias term.

2.3.2. Evaluation

When training a machine learning model, the process should always be evaluated. Therefore, after the training phase, there is a validation phase. In the beginning, we always make a split of the data. Often we do this split into an 80/20 ratio. Here 80% is used as training data where the model can use the labels and the features to train its parameters. After the training phase, it will use the other 20% of

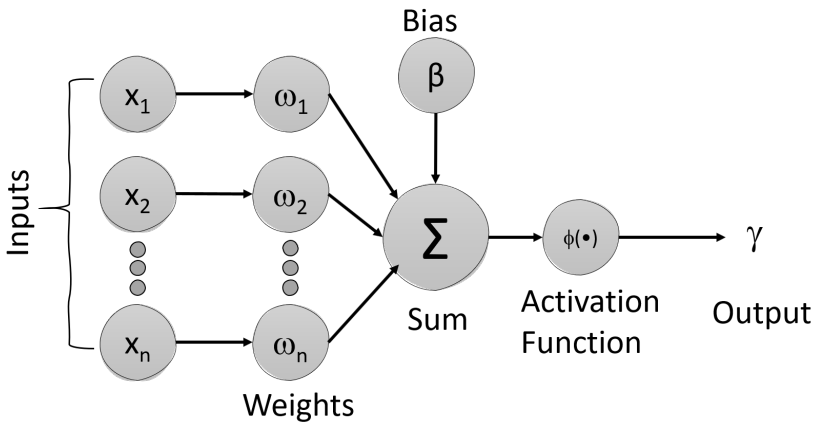


Figure 2.5: A zoom in view of an artificial neural network. In the figure is visualized how the output of a neuron is calculated.

the data, but it will not know the labels. So, the model will classify the features of the last 20% to the right labels. These labels are then compared with the original labels, and based on that comparison, we can calculate an accuracy based on the formula in Equation (2.3).

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{number of predictions}} \tag{2.3}$$

We have to do this split in training and validation data so that the model uses new data. It is hard to conclude something from a 100% accuracy when the model is evaluated on the same data as it was trained. However, when this is the case and even worse on its training data during the evaluation phase, the model has not learned the features from the data well enough. We briefly go into the two terms *overfitting* and *underfitting*. When a model is overfitted, it means the model cannot generalize and therefore fails to make correct predictions on the new data. The model has seen too much data similar to each other or has seen different objects too much. The model cannot interpret different changes to the features of objects it has not seen before and will not work well.
A model could also be underfitted during the training session. This means the data is too similar in the training set and cannot make reasonable distinctions between the different labels and corresponding features. It could also mean that the model has not seen enough data and has not learned well enough how to describe a label and which features correspond to these labels.

2.3.3. Activation Functions

One of the hyperparameters that are tuned in the model is the activation function. The output of the model is calculated by this function and will make a significant impact on the performance of the model. Without an activation function, the neuron's mathematical operation consists of dot products between the weights matrix and the input vector. This dot product will only result in linear formulas, and therefore the neuron will only be able to capture linear formulas. With the use of the activation function, the model can learn non-linear functions. In this thesis, we will use the following four activation functions. Therefore, we will briefly describe the own characteristics of each activation function here with the corresponding formulas.

The first activation function is the *TanH* activation function. This activation function limits the output between -1 and 1. The activation functions are non-linear in characteristics, and therefore we can stack layers when using this activation function. However, this activation function also suffers from the vanishing gradient problem. *TanH* is mostly used in recurrent neural networks. The formula of the *TanH* function can be seen in Equation (2.4).

$$TanH(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.4}$$

The second activation function is the rectified linear units (ReLU) function and is defined in Equation (2.5). This activation function returns a 0 for every negative value and returns the same value for

any other values, which makes this activation function cheap to compute. The activation function does not suffer from the vanish gradient problem, having many advantages; this activation function also has some disadvantages. Because the ReLU function is returning 0 for every negative value, there is a high chance that once a neuron is 0, it will never activate again. This is also known as the *dying ReLU* problem. This activation function is mostly not used in recurrent neural networks because the output is mapped linearly. In RNNs, the output can get remarkably big, and therefore using a ReLU function will make the value so big that it will be lost out of sight.

$$RELU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x >= 0 \end{cases} \tag{2.5}$$

The third activation function is the sigmoid activation function. This activation function maps every value from minus infinity to plus infinity to a value between 0 and 1. The formula of the sigmoid function can be found in Equation (2.6). There are some downsides to using the sigmoid function. For example, the exponential(e^x) function of the sigmoid function is computationally expensive. Moreover, this activation function also suffers from the vanish gradient problem.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

The last activation function that will be used in this thesis is the softmax activation function. The formula behind this can be seen in Equation (2.7). This activation function's logic is to turn the numbers that come as input into probabilities that sum to one. The output of this function is a vector that represents the probability distribution of the possible outcomes. This softmax activation function is mostly used for the last layer because it can classify the output to the different classes and gives the highest probability for the class that is most likely the correct label.

$$S(y_i) = \frac{e^{y_i}}{\Sigma e^{y_j}} \tag{2.7}$$

2.3.4. Weight Initializer

Another hyperparameter that is tweaked in a deep learning model is the weight initializer. With an optimal start value of the weights, the model will achieve the most in minimum time. Selecting the correct starting value for the weights will make it harder to get stuck in local minima by gradient descent. A weight initialization gives the starting value of the model and therefore influences the learning behavior. In this thesis, we will be using the following weight initializers:

LeCun uniform, draws samples from a uniform distribution within $[-x, x]$. This x value, or also called limit, is calculated in the following way:

$$Limit = \sqrt{\frac{3}{input}}. \tag{2.8}$$

The input listed in Equation (2.8) is equal to the number of input units in the weight tensor.

He uniform looks a lot like the previous discussed LeCun uniform. However, the number in the square root is now equal to 6. See Equation (2.9).

$$Limit = \sqrt{\frac{6}{input}} \tag{2.9}$$

Glorot uniform also draws samples from a uniform distribution. This initializer is the standard initializer of the model's kernel used in this thesis. The uniform distribution draws the values between two limits, which are calculated by the following formula

$$Limit = \sqrt{\frac{6}{input + output}}. \tag{2.10}$$

In Equation (2.10), the input is again the units in the weight tensor. The output is the number of output units in the weight tensor.

Uniform distribution is the most classic initializer. The values are drawn from a random distribution where the limits can be specified. By default, the values are drawn between -1 and 1.

2.3.5. Multilayer Perceptron

In this thesis, we use the multilayer perceptron (MLP). We do not design a new MLP, but we use already designed and described MLP models from related work; we only briefly discuss the multilayer perceptron's architecture. We will not dive deep into every layer and its function. This model consists of one input layer, at least a hidden layer, and an output layer. The MLP hidden layers can be configured with an infinite amount of neurons where the hidden layers are the layers that pass on the information from the input to the output layer. The neurons' size for the input layer should be equal to the number of features, where the neurons in the output layer are equal to the number of classes. The output from one layer to another is dependent on all the neurons in the previous layer. The output per neuron is calculated, and then, based on the activation function, a new output is calculated and fired to all the neurons in the next layer. The output of every neuron will be weighted; these weights can change per layer and per neuron. The network can learn the best configuration of weights for learning the input to the right output. MLPs are mostly used in stochastic research.

2.4. Deep Learning

Deep learning is a part of machine learning, which is most often based on artificial neural networks (ANN). In this thesis, we see many models that are part of two commonly used deep learning architectures. These architectures are the recurrent neural networks and convolutional neural networks. The most common use cases for deep learning techniques are computer vision, machine vision, speech recognition, natural language processing, audio recognition, machine translation, social network filtering, image restoration, and side-channel analysis.

Based on artificial neural networks, deep learning is inspired by the way the brain works and processes information. The way deep learning models can grow and extend their power has proven to be comparable or even surpass the human performance considering logic analysis. Their strength is finding and acting accordingly with massive datasets. Something that for a human would take ages to learn and find relations in it. When using deep learning, there is a constraint to call it 'deep' learning. This comes from using multiple layers inside a network. Using only one layer does not make it a deep learning network. The amount of layers is unbounded; however, every layer's size should be chosen and is therefore constrained.

In this thesis, we will use six commonly used deep learning models. This section explains these models and how their cells work, including which formulas are used to update the weights. The first section is a brief explanation of neurons and cells. In the second subsection, the recurrent neural networks are explained. In the third subsection is a brief explanation of the long short-term memory. In the fourth subsection, the gated recurrent unit is explained. After that, the following sections briefly discuss the convolutional neural network and autoencoder.

2.4.1. Recurrent Neural Networks

Recurrent Neural Networks (RNN) is a neural network that is called recurrent because it performs the same function on every input data and stores the output for the next input. This means the RNN stores the previous step input and merges that information with the current step input. With this feature, the network can have some kind of memory. The output of the neural network is thus dependent on past computation. Because the RNN has a memory, it is especially useful in training on sequential data. Sequential data is a stream of interdependent data. Sequential data could be, for example, the words in a sentence of a conversation or the data of a stock market. RNNs are also used for speech recognition.

An simple RNN is presented in Figure 2.6, with an input X_t, a hidden layer A and an output h_t. The formula for the current state can be written like

$$h_t = f(h_{t-1} + x_t), \tag{2.11}$$

where f is the activation function and for RNNs, usually the TanH activation function. The hidden

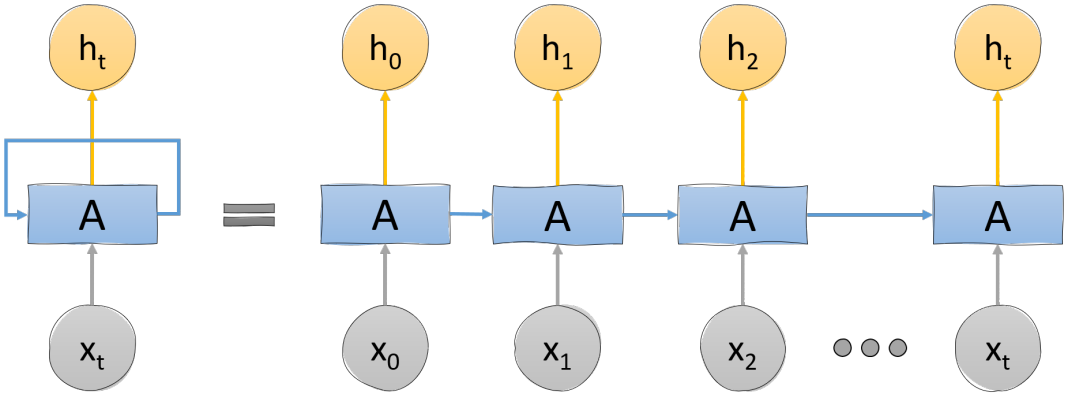


Figure 2.6: On the left side a visualization of an simple RNN. On the right side it is unrolled for better visualization.

state has weights for specific neurons. Say W_h are the weights of the current state, and W_{h-1} are the weights of the previous state. Then function (2.11) can be written as

$$h_t = tanh(W_{t-1} * h_{t-1} + W_h * x_t). \tag{2.12}$$

Considering a standard RNN, it has one major drawback. RNNs suffer from the vanishing gradient and exploding gradient problem. This makes the training of the neural network a challenging task. The vanishing gradient problem arises when there are multiple hidden layers. The data input has a wide range because an activation function, like the sigmoid or the TanH function, maps all the inputs to a small output. Therefore an enormous change in the input will give only a small change in the output. Therefore the gradient will be small. This problem does not arise with shallow networks, yet it does when more layers are used, and it results in the gradient being too small for practical training. A small gradient means that the layers' weight will not be updated as they should, which results in a significant inaccuracy of the network. The most straightforward solution given by related work is using the ReLU activation function. The second solution given by related work is to normalize the data before the activation part of the activation function.

A mathematical example of how this RNN works is as follows, take a sample input [0.1, 0.3, 0.6, 0.8]. The initial hidden state is commonly set to 0, and for the weight value, we chose a constant factor of 0.4 for every step, meaning we get the following Equation (2.13)

$$\begin{aligned}
h_0 &= TanH(W_0 * h_0 + W_0 * x_0),\\
h_0 &= TanH(0.4 * 0 + 0.4 * 0.1),\\
h_0 &= TanH(0.4),\\
h_0 &= 0.003998,\\
h_1 &= TanH(W_0 * h_0 + W_0 * x_1),\\
h_1 &= TanH(0.4 * 0.003998 + 0.4 * 0.3),\\
h_1 &= TanH(0.1215992),\\
h_1 &= 0.1210034,\\
h_2 &= TanH(W_1 * h_1 + W_1 * x_2),\\
h_2 &= TanH(0.4 * 0.1210034 + 0.4 * 0.6),\\
h_2 &= TanH(0.28840136),\\
h_2 &= 0.28066276,\\
h_3 &= TanH(W_2 * h_2 + W_2 * x_3),\\
h_3 &= TanH(0.4 * 0.28066276 + 0.4 * 0.8),\\
h_3 &= TanH(0.432265104),\\
h_3 &= 0.407212550.
\end{aligned} \tag{2.13}$$

As stated before, this is the most simple form of a sequential data model. In the following subsections, we present different variants of the RNN, including their mathematical formulas.

2.4.2. Long Short-Term Memory

The Long Short-Term Memory (LSTM) is a variant of the RNN that has been explained in the previous section. A so-called unrolled LSTM can be seen in Figure 2.7. As the name may already give away, the network can have a long memory, longer than the normal RNN has. This makes them more functional for data where the data's relationship is more spread over the complete trace. The LSTM was found in 1997 by Sepp Hochreiter and Jurger Schmidhuber [22]. One of the advantages of the LSTM network is that it deals with the previously discussed vanishing gradient problem. This is because LSTM cells have the option to let the gradient go unchanged to the next iteration. Therefore, the gradient will not vanish. Unfortunately, LSTMs are still vulnerable to the exploding gradient problem.

A simple LSTM cell looks as presented in Figure 2.7, with an input x_t. First the LSTM decides which information is important and what should be discarded. This decision is being made by the forget gate layer and is mathematically presented by Equation (2.14). The output is somewhere between 0 and 1 where 0 means forget and 1 remember the input x_t.

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f) \tag{2.14}$$

The next step in the LSTM is to update the cell state output. Equation (2.15) calculates which value has to be updated. Then new candidates are being calculated by Equation (2.16). These two formulas are combined which results in a new potential candidates for the cell state in the next LSTM cell.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.15}$$

$$\tilde{C_t} = TanH(W_c * [h_{t-1}, x_t] + b_C) \tag{2.16}$$

In this part the next cell state is actually calculated by Equation (2.17). This is a combination of the formulas seen before. Forgetting what we should forget and adding the new candidate values.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t} \tag{2.17}$$

The last step is to output the information and give this to the next cell state. This output is depended on the current cell state and the relevant information from previous cell and the input. So first is decided what is the relevant information. That is calculated in Equation (2.18), then this formula is used in combined with a TanH activation of the current cell state which results in Equation (2.19).

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o) \tag{2.18}$$

$$h_t = o_t * TanH(C_t) \tag{2.19}$$

2.4.3. Gated Recurrent Unit

The Gated Recurrent Unit (GRU), which is visualized in Figure 2.8, is introduced by Cho et al. in 2014 [12]. This unit aims to solve the vanishing gradient problem, which comes with the standard RNN. This updated version of the RNN looks and works similar to the LSTM. In most cases, the GRU and the LSTM perform equally. The GRU does this with his internal update and reset-gate. This gives the GRU two vectors which are trained to learn what to forget and what not. The first step in this process is the update gate. This update gate can be calculated with Equation (2.20). As we can see in the formula, the input is multiplied with the weight, and the output of the last cell is multiplied with the own weight. This update gate helps the model determine how much of the past information is relevant for the future and should be passed on.

$$z_t = \sigma(W^{(z)} * x_t + U^{(z)} * h_{t-1}) \tag{2.20}$$

After the update gate, there is also a reset gate in the GRU. This gate kind of resets the GRU and decides whether the past information should be forgotten or used for the current step. The output of the gate can be calculated using Equation (2.21).

$$r_t = \sigma(W^{(r)} * x_t + U^{(r)} * h_{t-1}) \tag{2.21}$$

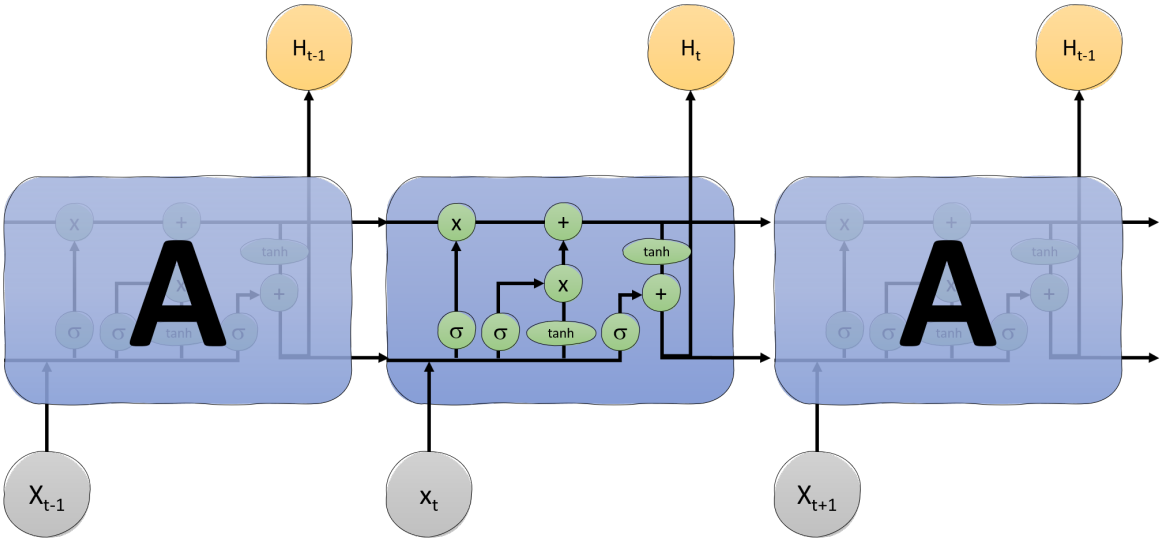


Figure 2.7: A grahpical visualization of a LSTM cell.

The formula looks the same as the one in the update gate. The difference is only in the weights that are used and where it comes back in the gate. Then we take the current memory of the GRU by using the formula written in Equation (2.22).

$$\bar{h}_t = tanh(W * x_t + r_t * U * h_{t-1}) \tag{2.22}$$

Here we store the current memory, which contains the relevant information from the past.

In the final step, we need to calculate the current unit's relevant information and what should be passed on to the next unit. Therefore we need the update gate together with the output of the current memory. The mathematical expression can be written as Equation (2.23).

$$h_t = z_t * h_{t_1} + (1 - z_t) * \bar{h}_t \tag{2.23}$$

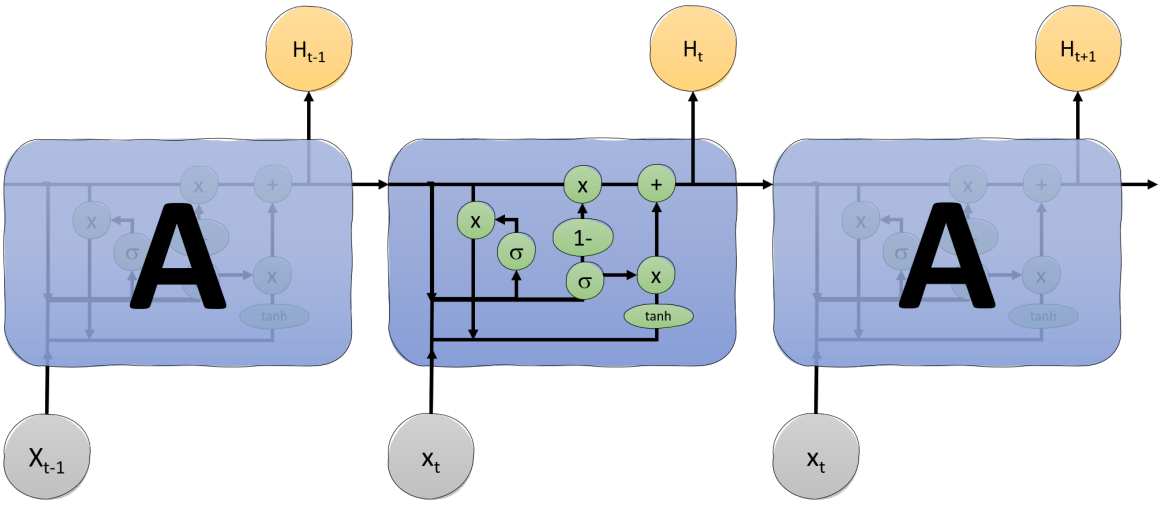


Figure 2.8: A graphical visualization of a GRU cell unrolled.

2.4.4. Convolutional Neural Network

In this thesis, we use the Convolutional Neural Network(CNN). We do not design new CNN models, but we use CNNs from related work, and here we discuss the general CNN architecture and specifics.

CNNs are mostly used in analyzing visual images. CNNs are most known because they are shift-invariant. They do have some overlap with the aforementioned MLPs but have a different approach towards regularization. The architecture of a CNN consists of an input layer, hidden layers, and an output layer. The power of the CNN is in these hidden layers. For a neural network to be 'deep' as used in deep learning, there should be multiple hidden layers. The hidden layers in CNNs are mostly convolutional layers that convolve with some multiplication. After the activation function, which is mostly a RELU activation, a pooling layer, then a fully connected layer, and a batch normalization layer are followed. The output layer will be a dense layer to make a classification into the right class.

2.4.5. Autoencoder

An autoencoder is a part of the family of artificial neural networks. The network tries to encode a value by using the encoder part of the network and then can decode the value by using a decoder to a value almost similar to the original value. An autoencoder is, by design, specialized in reducing and ignoring noise in data. An autoencoder consist of four parts:

- Encoder - Here, an encoded representation is constructed from the input. This happens by reducing the input dimension and then gives a compressed value of the input. The result is the encoded representation.
- Bottleneck - This is the middle of the autoencoder. Here are the values that are compressed by the encoder.
- Decoder - Here, the model learns how to reconstruct the original data from the encoded values. The results should be as close as possible as the original values.
- Reconstruction loss - Here is the evaluation performed. This part checks how much the output after the decoder looks like the original input.

In a side-channel analysis, an autoencoder can be used to clean traces before learning them. The autoencoder architecture is also used in natural language processing because of translating one language to another. An autoencoder is a particularity designed for many to many classification problems. However, every model can be used to build an autoencoder, as long as the encoder and decoder are doing the inverse of each other. We could, for example, use an LSTM as encoder and decoder, but also a convolutional layer as input and deconvolutional layer as output. The mathematical part of the autoencoder depends on the models used for encoding and decoding. To conclude, an autoencoder is a system of known models that reads the input, encodes it, decodes it, and then tries to recreate the output.

2.5. Natural Language Processing

The first natural language processing technique published was in 1950, published by a famous computer scientist Alan Turing [63]. In this publication, the writer tries to define a criterion of intelligence; something is now known as the Turing test. It is only till the late 60's when the first Natural Language Processing(NLP) systems are developed. When looking at NLP, we can briefly divide these into two subjects. Rule-based versus static NLP. Rule-based is by writing grammar for the language, which is done in the turning tests. The statical NLP, which became popular in the late 1980s, is based on machine learning techniques.

In this thesis, we mostly use recurrent models. These models have a massive use case in the natural language processing domain. The natural language processing domain is a subfield of artificial intelligence. The domain is mostly focusing on the interaction between human language and computers. The most common use case is for computers to understand the human language and learn the language with enormous datasets. This task is more challenging than it seems because a language is alive, and everything in a language could have more meanings. Therefore there are some specific natural language processing techniques. This domain has some specific techniques used a lot for recurrent networks, which are also commonly used with the human language.

In the following three subsections, we explain these natural language processing techniques. In the first subsection, the attention model is explained. The second subsection briefly discusses the

bidirectional layer. The last subsection explains the embedding layer and gives an example of how this layer works.

2.5.1. Attention Model

The attention model was first introduced by [5]. Attention is commonly used nowadays as extra features for sequential data models. A sequential model feeds a particular state's weight to the next state, creating one significant feature vector with all the previous words. This gives one big problem; features can vanish over time. Now the LSTM gives a solution to that problem, to make a direct connection from the beginning to the end of the cell. Therefore the model can pass one value immediately to the last part. However, when there are more exciting features of the sequence, what then? That is where attention takes place. Attention computes every single internal state like the model did before. However, then it creates an extra vector looking at the different options. Instead of a value that is being predicted at timestep ten, using an internal state vector generated by timestep nine and before. It can, for example, only look at the internal state at time steps three and five. So this attention vector that is generated can look closely at individual values in the sentence and can skip some values. The weights in the attention vector are between 0 and 1 and per vector has to sum up to 1. The context vector, which combines the attention vector and the internal state vector, is multiplied and then summed to make a context vector. This context vector, including the last internal state, is used for the next prediction and then concatenated, resulting in a final vector. This is helping the sequential model to look more specifically to the right part of the sequence. These attention weights are produced differently per model and can be tweaked by the user; in research, this is mostly called the alignment score. Examples of these scores are *Global attention, local attention, dot product, hard attention*. Mathematically we describe this as in Equation (2.24).

$$\begin{aligned} &\text{Internal states produces per CELL:} \\ &h_1, h_2, h_3, h_4.. \\ &\text{attention weights produces per cell:} \\ &e_1, e_2, e_3, e_4.. \\ contextVector_{t=5} &= h_1 * e_1 + h_2 * e_2 + h_3 * e_3 + h_4 * e_4 \\ concatVector_{t=5} &= [contextVector_{t=5} + h_4] \end{aligned} \tag{2.24}$$

2.5.2. Bidirectional Layer

The bidirectional layer is commonly used in NLP. When using this, we start to play with the time dependency. Like the word bidirectional already implies, the model learns in two ways. It connects two hidden layers of opposite direction to the same output. This results in a model that can learn the input from left to right and also from right to left. Meaning the model can look into the future before making a classification. So words that also depend on future words are better classified with this technique. However, in real-world scenarios, the use case is less convenient because we usually would not know what is coming next in some situations. For example, learning the stock market with bidirectional layers, knowing that a particular stock is decreasing in the future would make a huge impact, but that is not a real-life scenario. However, it could be used for handwriting and then recognizing letters in a word because it would be simpler to know which letter is before and after that particular letter.

2.5.3. Embedding

People refer to embedding as some kind of lookup table which is specific for the dataset. This is because an embedding layer also has weights that need to be learned for the particular dataset. When using an embedding layer, the input should be integer encoded. Embedding is defined as the first hidden layer of a model. We explain the function of an embedding with the following example. Take, for example, the following two sentences:
"This is my thesis"
" This thesis is good"
We then need to integer encode these two sentences, which will give the following two vectors. $[0, 1, 2, 3]$ and $[0, 3, 1, 4]$
Considering an embedding layer, we need to give two values when initializing. This is the input di-

mension and the output dimension. The input dimension should be equal to the number of distinct integers. In our example, that is 5, meaning we have five different words. Then we need to determine the output dimension. This means how big the output vector should be. In this example, we chose it to be equal to three. The embedding layer then creates a table which looks like Table 2.1. So here

Index	Embedding
0	[0.20, 0.60 , 0.40]
1	[0.35, 0.16 , 0.84]
2	[0.64, 0.78 , 0.09]
3	[0.88, 0.61 , 0.11]
4	[0.45, 0.54 , 0.10]

Table 2.1: Example of embedding matrix.

we see, the length of the table equals five, which is the input dimension and the embedding vectors have a length of three because we chose that as the output dimension. Changing the output dimension parameters means we can make the model more complex but also more specific. Now we have every word represented by a vector of size three instead of having a one-hot encoded vector of size five with 4 out of 5 elements as a zero. Also, it should be considered that this size scales linearly with the size of the dictionary. When using embedding, we can set our own embedding dimension. So not every word gets replaced by an embedding, but every unique word gets one embedding vector, which is used by looking up in the embedding table. The embedding values that are assigned to every word (or index) are also updated during training. Words that are commonly used together or that are holding a relation get the same values. Therefore we could learn a 'perfect' embedding space representing some kind of dictionary in a specific language. This example gives the useability of the embedding layer with words as an example. Since the side-channel analysis traces we use are float numbers, we adjust the embedding technique. This is described in Chapter 6.

2.6. Datasets

In this thesis, we use different public datasets. The reason for using different datasets is because they all have different characteristics that describe them. Those characteristics can be unprotected leakages, leakages with high noise, or leakages with random delay or masking countermeasures. In the side-channel community, most datasets that are used are the same. First of all, because capturing a dataset is precise work, much expensive equipment is a need, and even then, the measurement is sensitive to background noise. The second reason is a more practical trade-off; when everyone is using the same datasets, it is easier to compare the results of different models. We will use the following datasets in this thesis: DPAv4, ASCAD, CHES2009.

2.6.1. DPAv4

The DPAv4 dataset [1] is a dataset generated for the fourth DPA contest, which was organized by Telecom ParisTech and launched in July 2013. This thesis uses the first implementation where the traces are a masked implementation of AES-256 on an Atmel ATMega-163 smart card. The mask in this dataset is leaking first-order information [46]. Because of that, we can assume that we know the mask and therefore transform the implementation into an unprotected scenario. The dataset itself consists of 100 000 traces where every trace has 3 000 features. The leakage model of this DPAv4 dataset is described as

$$Y(k^*) = Sbox[P_0 \oplus k^*] \oplus M. \tag{2.25}$$

In Equation (2.25) M is the known mask and P the plaintext that is XORed with the round key k^*

2.6.2. CHES 2009

This dataset has a hiding countermeasure and is mostly referred to as the AESRD dataset[13]. The dataset is generated on an 8-bit Atmel AVR microcontroller, which was running an AES-128 implementation, with the random delay countermeasure? The dataset has 50 000 traces where every trace has

3 500 features. For this dataset, we attack the first key byte by attacking the first $Sbox$ operation. The leakage function is described in Equation (2.26).

$$Y(k^*) = Sbox[P_0 \oplus k^*] \tag{2.26}$$

2.6.3. ASCAD

The ANSSI SCA Database (ASCAD) [4] is a database that provides a benchmark for the side-channel community. The authors try to start a similar database as the MNIST database for figures, but for the side-channel analysis community. The ASCAD database is generated in July 2018 and has been introduced in [56]. The database has 50 000 profiling traces and 10 000 attack traces. These traces are measured and captured from electromagnetic emanation. The target platform has an 8-bit AVR microcontroller, which has a running implementation of a masked AES-128. The traces captured from this ATMega8515 are raw added to the database. Likewise, in the MNIST database, there is a preselected window equal to 700 features for the leakages of the third subkey byte. In this thesis, we use the same window as the authors set for the database. The leakage function of the masked ASCAD database is in Equation (2.27).

$$Y(k^*) = Sbox[P_2 \oplus k^*] \oplus M \tag{2.27}$$

In Equation (2.27) M is the known mask and P the plaintext at position three that is XORed with the round key k^*.

3

Related work

This chapter discusses the related work for this thesis. We take a look at relevant publications in the domain of profiled side-channel analysis. We show what is done within the side-channel analysis and where this thesis fits in the domain. Finally, we look at different recurrent neural network publications to know state of the art in sequential data models. By pointing out the current research, we can conclude what is missing and formulate three different research questions for this thesis with a few sub-questions.

In the first section, we take a look at machine learning in the side-channel analysis. The second section looks at the deep learning models in side-channel analysis. After this, the third subsection goes into the recurrent neural networks and is also part of the deep learning family; however, it is not yet used as a suitable side-channel model. Therefore this section briefly discusses the usability of RNN in other domains. The last section goes into natural language processing techniques, commonly used together with RNN models. Looking in the last section at what is state of the art and what kind of use cases are these techniques used nowadays.

3.1. Machine Learning in Side-Channel Analysis

Side-channel analysis is a powerful way to break encryption using techniques with big data analysis. Therefore the strength of an attack and the danger of a side-channel attack goes hand in hand with the data analytic model. The first published side-channel attack was in 1996 by [32]. This attack was the start point for side-channel analysis; after that, they started to use different attacks, which are Differential power analysis [31] (1999), Template attacks [10] (2003) and Stochastic approaches by [59] (2005). From there on, they made a jump to use Machine learning techniques. The first side-channel attack using machine learning techniques was by [24] in 2011, there they used the least square support vector machine (LS-SVM). From that point forward, people started to use machine learning techniques for breaking complex cryptography algorithms. They started to use Random Forest [20, 37] and Support Vector Machines [19, 53] to break protected and unprotected implementations. They showed the world the power of side-channel analysis made designers of hardware more aware of the risks. Therefore there are more countermeasures of both categories present in hardware implementations nowadays [44]. Unfortunately, these countermeasures are also breaking for the strength of the machine learning models [55]; nevertheless, they do not make it easier. The complete overview was published in a survey on machine learning techniques in 2020 [18].

In [62] the authors propose a unified framework proposed to evaluate side-channel analysis, which is based on the guessing entropy and success rate. This is because the authors argue that accuracy is not the only way to evaluate the correct key. In the end, the goal is to break the encryption and not to have a correct intermediate value. In [52] it is proven that accuracy is not the most suitable option when performing a side-channel analysis attack with machine learning, but guessing entropy should be used. This gives more insight into the model's performance instead of the accuracy; there are even examples of models with terrible accuracy but with a good guessing entropy. When using guessing entropy, the order of traces does influence the effectiveness of breaking the encryption. Therefore guessing entropy has to be done with a random permutation of the attack traces multiple times and then average the result.

3.2. Deep Learning in Side-Channel Analysis

After machine learning showed strength in side-channel analysis, the need came to improve the attacks. Where hardware became cheaper and more secure, the complexity of models became bigger. Also, countermeasures were a reason for the community to start looking for more complex models. From then on, the side-channel domain started to look into deep learning models for side-channel analysis. The first model used were multilayer perceptron by [21, 52, 56] and after that also convolutional neural networks [38, 56, 60]. After those publications, the whole side-channel domain went looking for better models in the deep learning domain. Deep learning is a powerful type of machine learning, so it made some sense that it had better attack results than the previously tried machine learning models. Deep learning also showed their strength with image recognition, which also, just like side-channel analysis, has to deal with high dimensional data. Therefore the transfer to deep learning was a logical step.

In [56], a side-channel dataset is published, called the ASCAD dataset. In his work, extensive research is done by analyzing different deep learning models and these networks' effectiveness on the newly acquired dataset. The research is mainly focused on finding the best hyperparameters for the deep learning models, something that is quite a popular strategy for researching side-channel analysis. At the moment, the community is trying every possible hyperparameter for convolutional neural networks, which resulted in a saturation of the results. Scientists and researchers are competing by how many traces they need to reach a guessing entropy of zero, and then a new publication shows that some other model is half the size but has similar results [74]. After successful results with current state-of-the-art only minor changes are introduced, while other potential algorithms are not explored. In this thesis, new interest models will be explored.

As mentioned before, researches are mainly focusing on CNNs now for deep learning models when performing a side-channel attack. These have mainly two reasons, which are two valuable strengths for a CNN in general. The first reason is that CNNs are spatial invariant; this means that the place of a feature does not matter for CNN to recognize it. This concludes that a hiding countermeasure will be a bad countermeasure when using a CNN because the model will still be able to extract the features independent of the place in the feature vector [9]. The second big advantage of CNNs is that they can extract the essential features without any preprocessing method. That means that other, maybe even more complicated, preprocessing techniques can be neglected, and then a researcher can only focus on the performance and the hyperparameters of the CNN [29]. Moreover, the author shows that it is hard to choose a uniform model for attacking all the different datasets. This means CNN is still really sensitive for different datasets, and for every other dataset, the architecture has to be designed again.

3.3. Recurrent Neural Network

While the side-channel community has much evidence that machine learning models and mostly the deep learning techniques from the machine learning models are quite impressive, the focus has never really been on recurrent neural networks. These neural networks are part of the deep learning network and therefore feel like the community forgets them, or the community is making a linear approach with CNNs without iterating to look for something else. To the best of our knowledge, the first scientific publication using RNNs for side-channel analysis is by Maghrebi et al. in [39]. They use different deep learning techniques to attack DPA Contest v2, where they only use an LSTM for the recurrent family models. The model converges after 1 000 attack traces. The reason they give is that the leakage of the hardware is not time-dependent. Furthermore, they attack a first-order masked AES implementation. However, in the plot with results is no LSTM model. This could mean the results were so terrible that they chose not to show it. There are no other scientific publications for recurrent neural networks in side-channel analysis. This means the previously mentioned paper has set a baseline, but then no further investigation occurred, meaning there is a gap of knowledge for the recurrent neural networks in the side-channel domain. Why are these networks not explored any further for side-channel analysis? Can they be an excellent alternative deep learning model for side-channel analysis? What is the strength of these models?

The family of recurrent neural networks, also called sequential data models in this thesis, is bigger than only the RNN, as it also includes the GRU and LSTM. While the LSTM is commonly used in machine learning, the use of the GRU model is not very common. Considering LSTM, they're used many times in EEG classification [34, 48]. There the authors use LSTM networks to classify EEG brain waves and find diseases. An interesting aspect of publications using recurrent networks is that they

always use additional methods for the best optimization. For example, in [68], they combine an LSTM with a CNN to get the best results. Then there is [67], where the author proposes an LSTM model with a linear regression model. Concluding from these publications, the question arises, what special method should and could be used for the sequential data models that will be used in this thesis that work particularly well with side-channel analysis?

3.4. Natural Language Processing techniques

In this thesis, we will look into preprocessing and cleaning techniques for the traces as input. The reason is that research publications argue that autoencoder is better than recurrent neural networks [28]. Especially the use of attention between RNNs [26]. After many improvements in different attention models, Google came with a publication where they only used attention instead of weights [66]. Here the feedforward of values is only attention instead of weights. The paper is highly cited and a breakthrough for the natural language processing domain. They mostly used sequential data models to translate one language to another; they showed the most effective way was with a transformer model. This model works like an autoencoder and uses the attention as input and output of the autoencoder. The use of autoencoder is explored as a preprocessing technique for side-channel analysis by [72]. When considering preprocessing techniques for side-channel analysis, we mostly see three different approaches:

- converting the dataset into another representation,
- Reduce the amount of noise in the trace by a preprocessing technique, or
- An alignment method to make sure the measurement is aligned.

Considering the autoencoder with translation and the previously named publication, we are mostly interested in the second option, where we reduce the amount of noise in a trace by preprocessing. Most publications do not name any preprocessing method at all. With this, the questions arise, why the preprocessing methods are not used? What kind of preprocessing methods are there, and which ones are suitable for removing noise in the side-channel analysis?

3.5. Research questions

Considering the related work mentioned above, we here declare the research questions are addressed in this thesis. These research questions are more precisely formulated questions than the ideas which were given in section 1.1. The precisely formulated research questions that are addressed in this thesis are:

- In Chapter 4 the following research question is covered:
  RQ 1. How could sequential data models be used for side-channel attacks?
  SubRQ 1.1 What are good hyperparameters for sequential data models when dealing with side-channel analysis?
  SubRQ 1.2 What is the effect of linear regression preprocessing technique when used before applying long short-term memory when dealing with side-channel analysis?

- In Chapter 5, the following research questions will be researched:
  RQ 2. Can an LSTM autoencoder be used to de-noise a trace and make a better side-channel attack in terms of guessing entropy function?
  Sub RQ 2.1 How good do the denoised traces fit on the original traces?
  Sub RQ 2.2 How do these denoised traces perform compared to original traces using CNN to evaluate the performance?

- In Chapter 6 the following research question will be researched:
  RQ 3 Should embedding be used as a preprocessing method in side-channel analysis?
  Sub RQ 3.1 What output dimension should be used when using an embedding layer?

The research questions in Chapter 4 will give us more insight if more deep learning techniques should be used in side-channel analysis. We are mostly searching for the best sequential data model and what optimization is useful when using these recurrent models. Then chapter 5 will give us more insight if attention is a suitable option for an LSTM model when denoising a trace. We are interested in a translation model that can denoise a trace in the same way a model translates one sentence to another trace. Chapter 6 finds it is strength in using a specific technique for recurrent models and see their value for side-channel analysis.

4

Evaluation of Sequential Data Models

In this chapter, different setups of RNN, LSTM, and GRU are evaluated in side-channel analysis. In the first section, the methodology for this chapter is explained. In the second section, the original DPAv4 dataset is used, and we explore different hyperparameters and sequence length. We want to know the influence of different sequence lengths on the sequential data models. In the third section, a preprocessing method is used as described in [69]. The technique reduces sequence length using linear regression, which has not been used before in side-channel analysis. In the fourth section, the bidirectional layer is investigated. Because till then, the results have not been overwhelming compared to CNN results. Therefore we introduce a new layer that is typical for sequential data models. The last section looks at different datasets and leakage models to better understand how sequential data models perform for the side-channel analysis domain. This chapter has two goals. The first goal is to find the best hyperparameters for using these sequential data models in side-channel data. The second goal is to give well-substantiated advice if these models should be used in the side channel domain and clearly explain why they should or should not be used. The chapter ends with a conclusion section that summarizes all the conclusions found in this chapter.

4.1. Methodology

All the experiments in this chapter use the same methodology. First, we do a grid search for the best hyperparameters values that should be used for the models. The best hyperparameter search is essential for making a right and well-informed decision about sequential data models in side-channel analysis. From related work, we find what common values are used for the hyperparameters. We can evaluate a side-channel attack with a guessing entropy plot. Here we plot the guessing entropy value on the Y-axis and the amount of traces on the X-axis. We have used different setups of hyperparameters and compared the guessing entropy plots. Based on which model reaches a guessing entropy of 0, the earliest (with the least amount of traces needed) performs better. For the hyperparameter's effect on the model's performance, we explored a big grid with different values. Then, we fix all the values but only differ one hyperparameter. We can see then how the parameter that is differing influences the results in the guessing entropy plot [40]. For the experiment's correctness, every attempt is being run ten times with the same values for the hyperparameter.

The setup for all the experiments is as follows: The model is trained in a simple form, meaning without any extra preprocessing and normalizations inside the model [27, 35]. The only preprocessing that is happening is a batch normalization before the model is trained. A sequential data model only accepts three-dimensional inputs; however, the dataset's input is two dimensional. Therefore the dataset is transformed into a three-dimensional dataset. This means the x-dimension in the dataset is a trace, the y-dimension is a timestep, and in the z-dimension is the value of that timestep. In this way, the information that holds the new dataset has three dimensions. The dataset is holding the same information as before but now compatible with a sequential data model. There could even be more information per timestep in this way; unfortunately, this information is not present in the currently available datasets. Furthermore, we use the categorical cross-entropy function as the loss function. The optimizer used in this whole chapter is the adam function with a fixed learning rate of 0.001.

4.2. RNN, LSTM, and GRU

A full grid search for the best hyperparameters for the RNN, LSTM, and GRU is performed and evaluated in this section. We only used the DPAv4 dataset to find the best hyperparameters for the simplicity of the grid search.

The hyperparameters tested in the grid search experiments are batch size, amount of units, dropout, recurrent dropout, activation function, weight initializer, training size, and different amount of layers. All the hyperparameters with different values that are explored can be seen in Table 4.1.

model	hyperparameters	values
RNN, LSTM, GRU	*batch size*	1/30, 1/15
	units	1, 16, 32, 64, 100
	dropout	0.0, 0.2
	recurrent dropout	0.0, 0.2
	weight initializer	Lecun uniform, He uniform, Glorot uniform, Random Uniform
	training size	8 000, 2 0000, 4 0000
	layers	1, 2, 3
	activation function	ReLU, TanH, sigmoid

Table 4.1: The different evaluated hyperparameters and corresponding values in the different experiments.

This paragraph is an explanation of the chosen values for the hyperparameters. The batch size is divided into parts of the sequence length. For example, with a batch size of 1/30, the batch size is one when the sequence length is 30, and ten when the sequence length is 300. Using this technique, the experiments are still comparable when the sequence length differs between them. The experiments in literature have the same batch size compared to the sequence length in [45]. More importantly, the module of the sequence length and the batch size should be a real integer. Otherwise, the last batch of a trace has a different length than the batches before. In certain cases, this could happen, then the last part of the sequence length is neglected. According to [58], the number of layers used should be between 1 and 3. Where 3 is a complex network, and one is shallow but sufficient to capture the sequential dependencies. The units used in a cell should be the same as the number of data points at a particular time step. In the case of DPAv4, we only have one value; this means that the number of units should be equal to 1. However, it is not yet proven that more units work better. In other research, it is most time advised to use the same amount of units as features. The amount of units is determined arbitrary and is a multiplication of a byte. The different weight initializers that are used in this experiment are found in [56]. That paper is another well-cited paper researching the best weight initializer for CNN using the ASCAD dataset. The different weight initializers should give insight into the consistency of the model. The dropout and recurrent dropout values are adjusted to see the difference. However, according to [69], dropout and recurrent dropout does not do much with the classification of the model. The last hyperparameter that is tweaked is the activation function used in the model. Mostly the TanH activation function is used in sequential data models because of the vanishing gradient problem. However, in some specific cases, we saw a difference when using different activation functions. Therefore we used the three most common ones used in related research and are compatible with recurrent layers.

4.2.1. DPAv4 with Sequence Length of 3000

The sequential data model should capture and remember the leakage of the model and classify it to the right intermediate value. In the setup of this experiment, the traces had a length of 3 000 values. Meaning that there are 3 000 timesteps in a trace, and every time step has one value, the leakage of the chip at that particular timestep. The number of times the correct key byte is at position 0 after guessing entropy can be seen in Table 4.2.

In Table 4.2, are the results from the experiment with a sequence length of 3 000. A first glimpse on the table proves the point made before; the best results are made when the amount of data holding at each timestep is equal to the number of units in the model's cell. Therefore we used one unit per cell in the rest of the experiment. Secondly, there are some "X" marked in the tables. This is because these configurations took more than 48 hours to be trained and evaluated. Therefore these experiments have never been finished and neglected for this table. In these experiments, the batch size and amount of

	LSTM				RNN				GRU			
1 Layer		Bs	1/30	1/15		Bs	1/30	1/15		Bs	1/30	1/15
	Units	D\RD	0	0	Units	D\RD	0	0	Units	D\RD	0	0
	1	0	**10**	**10**	1	0	**10**	**10**	1	0	**10**	**10**
	16	0	8	7	16	0	9	7	16	0	0	0
	32	0	0	2	32	0	5	0	32	0	0	2
	64	0	5	0	64	0	4	5	64	0	0	1
	100	0	6	0	100	0	8	9	100	0	1	2
	1	0.2	**10**	**10**	1	0.2	**10**	**10**	1	0.2	**10**	**10**
	16	0.2	9	**10**	16	0.2	0	8	16	0.2	**10**	8
	32	0.2	0	4	32	0.2	**10**	**10**	32	0.2	0	2
	64	0.2	2	6	64	0.2	8	**10**	64	0.2	1	1
	100	0.2	2	5	100	0.2	8	9	100	0.2	2	0
2 Layers		Bs	1/30	1/15		Bs	1/30	1/15		Bs	1/30	1/15
	Units	D\RD	0	0	Units	D\RD	0	0	Units	D\RD	0	0
	1	0	**10**	**10**	1	0	**10**	**10**	1	0	**10**	**10**
	16	0	**10**	2	16	0	0	8	16	0	5	1
	32	0	0	3	32	0	0	4	32	0	0	0
	64	0	0	1	64	0	1	1	64	0	0	2
	100	0	0	6	100	0	0	2	100	0	0	2
	1	0.2	**10**	**10**	1	0.2	**10**	**10**	1	0.2	**10**	**10**
	16	0.2	2	5	16	0.2	2	2	16	0.2	3	6
	32	0.2	9	**10**	32	0.2	**10**	8	32	0.2	0	3
	64	0.2	4	**10**	64	0.2	4	5	64	0.2	0	3
	100	0.2	3	1	100	0.2	2	4	100	0.2	0	0
3 Layers		Bs	1/30	1/15		Bs	1/30	1/15		Bs	1/30	1/15
	Units	D\RD	0	0	Units	D\RD	0	0	Units	D\RD	0	0
	1	0	**10**	**10**	1	0	**10**	**10**	1	0	**10**	**10**
	16	0	3	3	16	0	0	3	16	0	4	7
	32	0	7	0	32	0	0	0	32	0	1	2
	64	0	2	6	64	0	1	4	64	0	1	0
	100	0	2	0	100	0	2	0	100	0	0	0
	1	0.2	**10**	**10**	1	0.2	**10**	**10**	1	0.2	**10**	**10**
	16	0.2	1	3	16	0.2	5	3	16	0.2	5	**10**
	32	0.2	0	3	32	0.2	7	7	32	0.2	0	0
	64	0.2	0	0	64	0.2	8	8	64	0.2	0	0
	100	0.2	0	X	100	0.2	5	X	100	0.2	0	X

Table 4.2: Results of the experiment with 3 000 values, the value in the cell represents the amount of time the model was able to find the correct intermediate value during the validation phase. The rows represent 1,2 and 3 layers and after that split into different units and different dropout value(D). The columns represent the three different sequential data models and after that split in different batch size (bs) and recurrent dropout (RD).

units were large, meaning more parameters to train, which would take more time to finish. However, we do not expect the cells that now hold an X to have a surprisingly high intermediate value score. Lastly, we see some other cells holding 10 out of 10 correct intermediate values. These values are neglected because there is no consistency between the number of layers and the number of used units.

The time of a model to learn and perform an attack on a set of traces is not a critical factor in the side-channel analysis. The attack performed is a profiled attack where the attacker already has the traces and can learn a suitable model in profiling. However, looking at different studies, a very complex network can break the key within 48 hours of training [74]. When using recurrent dropout, the network cannot learn a model within these 48 hours, and therefore these values are missing in the table. The reason for this long computation time is that no GPU speedup is possible because the value of t+1 depends on the value calculated at time t. Because of this, no parallel computation is possible. This could mean that when the sequence length is shorter, the model's learning time could reduce, and then recurrent dropout could be used. However, in this setting, with a sequence length of 3 000, we

neglected the recurrent dropout of 0.2.
The table only shows the number of times the model is finding the correct intermediate value. It shows that every sequential data model with one unit, independent of layers or batch size, can find the intermediate value 10 out of 10 times. However, it does not give any information about which model is better than the other, which should be concluded by looking at the guessing entropy. From this table, we can conclude that the chosen layers and batch size modules are suitable options for experimenting with the best hyperparameter. That we should not explore the number of units any further and that the dropout and recurrent dropout are also well-picked values.

First, the number of layers has been considered; see Figure 4.1. Here the mean is taken from the ten experiments when considering the amount of traces for guessing entropy to reach zero. Furthermore, the mean is taken from the different setups of variables to get an average on a layer-based performance and on the amount of traces needed to find the correct intermediate value. What can be seen from this figure is that using more layers increases the performance considering the LSTM and RNN model. However, the GRU is performing worse compared to the other two models and to itself regarding more layers. From this experiment, we can conclude that using more layers with RNN and LSTM increases the performance slightly when a trace consists of 3 000 values.

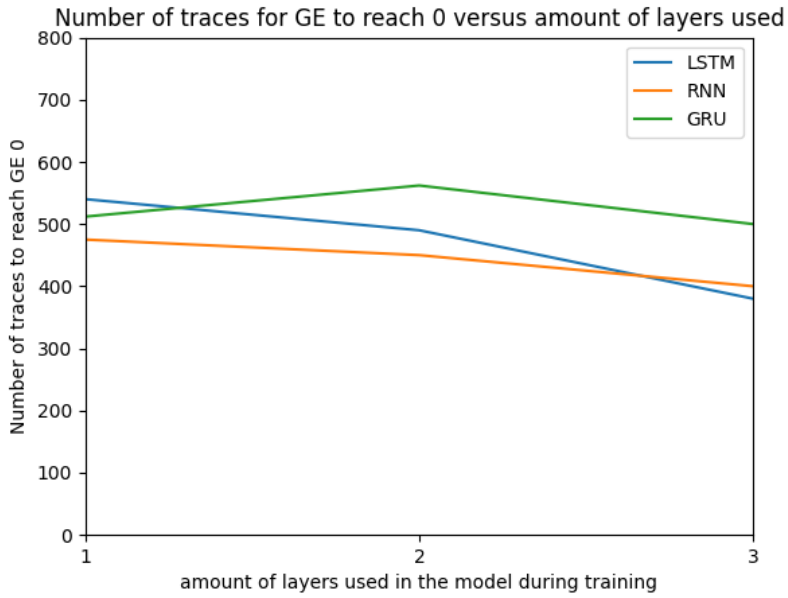


Figure 4.1: On the Y-axis the number of traces needed to reach a guessing entropy of 0 on the X-axis the number of layers in the different models. All the experiments are combined by taking the mean of the different results.

We see that most models reach a GE of zero after around 800 traces. The results of this experiment should be compared with the state of the art [70]. In those researches, they can break the cipher in the DPAv4 dataset within ten traces. Comparing those results with these results makes clear that at the moment, there should be a significant improvement in the sequential data models to win the competition of state of the art. As explained in the background, sequential data models are used in text and speech recognition. In those cases, a sequence has a length of, for example, a sentence, 20. Also, current research in [73] states that the maximum length a sequential data model could capture is around the length of 1 000. We, therefore, reduced the sequence length in this research.

4.2.2. DPAv4 Selected Time Window of Size 450

In this subsection, the sequence length is reduced. As stated in the previous section, the results were not impressive enough to take it to the next step and do a full grid search. One of the reasons we want to reduce the sequence length is that the results are too far from the state-of-the-art attacks; the other reason is time complexity. For this experiment, the sequence length is reduced to 450 timesteps. These 450 timesteps are taken from timestep 1750 to timestep 2200. This is because, according to [65], there

are the most distinctive features that a CNN used to learn and capture the traces. Noteworthy is that we want the leakage values to be sequential. Therefore only one window size is chosen instead of selecting multiple smaller windows of a different part of the trace with higher leakage. Furthermore, only one unit per cell is used because of the previous experiment, and the dimension is not different.

	LSTM				RNN				GRU			
1 Layer		Bs	1/15	1/30		Bs	1/15	1/30		Bs	1/15	1/30
	Units	D\Rd	0	0	Units	D\Rd	0	0	Units	D\Rd	0	0
	1	0	**10**	**9**	1	0	**10**	**10**	1	0	**10**	**10**
	1	0.2	**10**	**10**	1	0.2	**10**	**10**	1	0.2	**10**	**10**
2 Layers		Bs	1/15	1/30		Bs	1/15	1/30		Bs	1/15	1/30
	Units	D\Rd	0	0	Units	D\Rd	0	0	Units	D\Rd	0	0
	1	0	**10**	**10**	1	0	**10**	**10**	1	0	**10**	**10**
	1	0.2	**10**	**10**	1	0.2	**10**	**10**	1	0.2	**10**	**9**
3 Layers		Bs	1/15	1/30		Bs	1/15	1/30		Bs	1/15	1/30
	Units	D\Rd	0	0	Units	D\Rd	0	0	Units	D\Rd	0	0
	1	0	**10**	**10**	1	0	**10**	**10**	1	0	**10**	**10**
	1	0.2	**10**	**10**	1	0.2	**10**	**10**	1	0.2	**10**	**10**

Table 4.3: Results of the experiment with a sequence length of 450. The value in the cells represents the number of times a model was able to find the correct intermediate value.

In Table 4.3, the results of the experiment are presented. Here can be seen that in almost every experiment, the model can learn and correctly predict the intermediate value. Only in two cases, it predicts nine out of ten, which should be observed is that in those two cases, it predicted the correct intermediate value, not a key rank zero but at key rank one. Therefore, we conclude that the window size is chosen correctly but wonder if the model is stable enough. From this, we can expect that the experiment is not consistent enough. Because of that, we used different input sizes to see if the model is sensitive to the dataset size. Also, we want to see if the model has been over- or under-fitted.

In this experiment, we differentiated the training size and, therefore, also the validation size. The validation size is in this thesis 20% of the training size. Just to clarify, the different sizes for the dataset used to work with are 8 000, 20 000, and 40 000 traces, which means respectively a training size of 6 000, 16 000, and 32 000. The sizes to train with have been taken by looking at other research. The guessing entropy of all the experiments showed using one layer is working way better than two and three. Therefore Figure 4.2 is only with one layer and for RNN, LSTM, and GRU. The mean is taken from the ten experiments. Then, we get four different GE's (for dropout 0.0 and 0.2, recurrent dropout set to 0, units set to 1, and batch size set to 1/30 and 1/15). These four are also summed and divided by four, giving an average guessing entropy for different hyperparameters' values. These are combined and plotted in Figure 4.2 to compare different training size.

According to Figure 4.2, it is not possible to draw any positive results compared to other side-channel analysis research. What can be seen is that the RNN is not performing any better when the dataset size is increased. For the LSTM model, there is no consistency in finding the key. At first, it looks the model performs even worse, but then guessing entropy starts to decline to result in around an even good result than with the smaller data set. Concluding this experiment shows that dataset size does not influence the model's effectiveness in the side-channel context. Moreover, it shows that the best results are found when the data set size is 8 000. Therefore we kept using that size for the experiments. The GRU model increases over time, which could mean the model's overfitting when using more traces in the training and validation phases.

There is no improvement in the consistency after changing the data set size. Therefore, this experiment looked into improving the consistency of the model. We expect to see some improvement by using different weight initializers [3, 49]. For this experiment, we use the following weight initializers; *Random Uniform, Glorot Uniform, He Uniform, and LeCun uniform*. These are also the weight initializers that are used in [57]. Because in the previous experiment, we saw only the LSTM not being consistent (rising and then dropping), this experiment only used the LSTM with one layer. Furthermore, we are using the same setup as previously discussed with one unit per cell, dropout of 0 and 0.2, and recurrent dropout of 0. Batch size equal to 1/15 and 1/30 and then taking the mean of the four values to average the performance of the weight initializers inside the layer over different setups. When us-

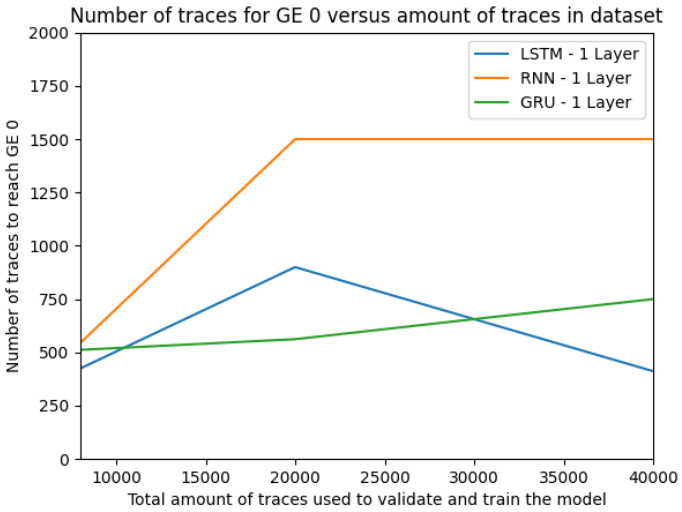


Figure 4.2: On the Y-axis the amount of traces needed to reach a guessing entropy of 0 on the X-axis the size of the dataset used for the model to train and validate on.

ing different weight initializers, all the experiments are more consistent because they reach the right intermediate value within, on average, the same amount of traces as seen before, so there is no performance decrease. Also, every individual experiment reaches the right intermediate value, which was not always the case in the previous experiments. All these conclusions can be seen in Table 4.4. In Figure 4.3, there is a visualization of the means of the guessing entropy for the different experiments. What can be seen is that He Uniform as weight initializer is achieving the best-built, because He Uniform can get a key rank of zero after around 350 traces, where other weight initializers need at least 500 or 550 traces. Looking at the same setup where the random weight initializers are used, the experiments show that there is a big difference between Random Uniform and He uniform weight initializer. At least we can conclude that using the other named weight initializers decreases the performance of the model. Furthermore, He Uniform and random weight initializer show good results considering their performance on the sequential data models. However, until now, we see more consistent results with the He Uniform weight initializer. Because with the random weight initializer, we saw the experiment having a guessing entropy which is worse compared to the He Uniform weight initializer. Therefore we used the He uniform weight initializer from now on in all the experiments.

weight initializers	Successful experiments out of 10	GE of experiments on average over all different setups
Random Uniform	**10**	550
Glorot Uniform	**10**	500
He Uniform	**10**	350
LeCun Uniform	**10**	600

Table 4.4: Overview of results of different weight initializer and their corresponding guessing entropy.

We further investigated the best activation function to use in the sequential data model. The standard activation function used by Keras and Tensorflow is the TanH activation function. TanH and sigmoid are mostly preferred when a model needs to learn complex dependencies. This is because the activation functions are non-linear and, therefore, better in capturing these complex dependencies. At first, ReLU does not seems to be a suitable option. This is because the linearity and the values in the cells do explode over time. However, according to [36], a ReLU activation function can give some extra stability to the network. Therefore we used the previously discussed activation function in the following experiment. The setup is the same as discussed before. The results are visible in Figure 4.4.

In Figure 4.4 are the different activation functions with the models. The ReLU activation function was

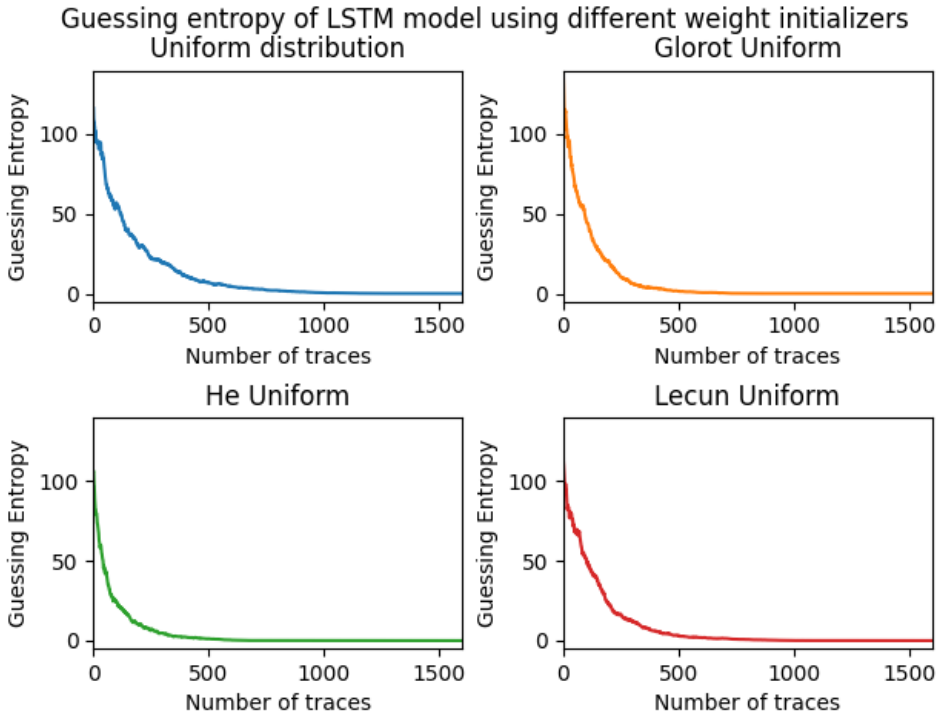


Figure 4.3: On the X-axis, the amount of traces needed to reach a guessing entropy of 0. On the Y-axis, the guessing entropy value corresponding to the amount of traces. In every picture is the other weight initializer used, the used initializer is written above every subplot.

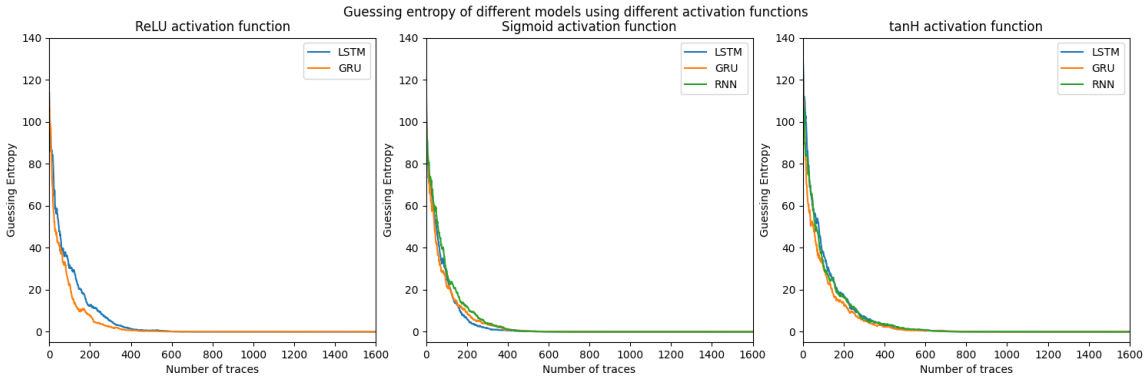


Figure 4.4: On the X-axis the amount of traces needed to reach a guessing entropy of zero. On the Y-axis the guessing entropy value corresponding to the amount of traces. In every figure is a other activation function used, the used function is written above every subplot.

not able to learn anything when the RNN model was used. Therefore this model is not present in the left plot. Furthermore, we see that all three models respond the same way to an activation function. This means that the results' difference is minimal for each sequential data model, looking at every activation function separately. Comparing the three results, we see that the TanH activation function performs the worst compared to sigmoid and ReLU, which is interesting because the TanH is the standard activation function used for sequential data models in Keras. However, this could be explained by the type of data used in this thesis, which is side-channel leakages. The ReLU was unstable with RNN; therefore, we chose to keep using the sigmoid activation function. It also seems that sigmoid giving slightly better results, however not significant compared to the ReLU activation function.

The last three hyperparameters that need to be explored are batch size, dropout, and recurrent dropout. The batch size determines how many times the parameters of the cell should be updated. The

dropout means how much of the current values should be kept and how much should be dropped; this prevents the model from overfitting. Recurrent dropout is the same as a dropout in terms of meaning, but the dropout is applicable for the weights passed through the next value, and recurrent dropout is to the cell's inner state. In this experiment, we run the different combinations of setups per model. For every model we plot the different hyperparameters and compare the differences we see in Figure 4.5 and Figure 4.6.

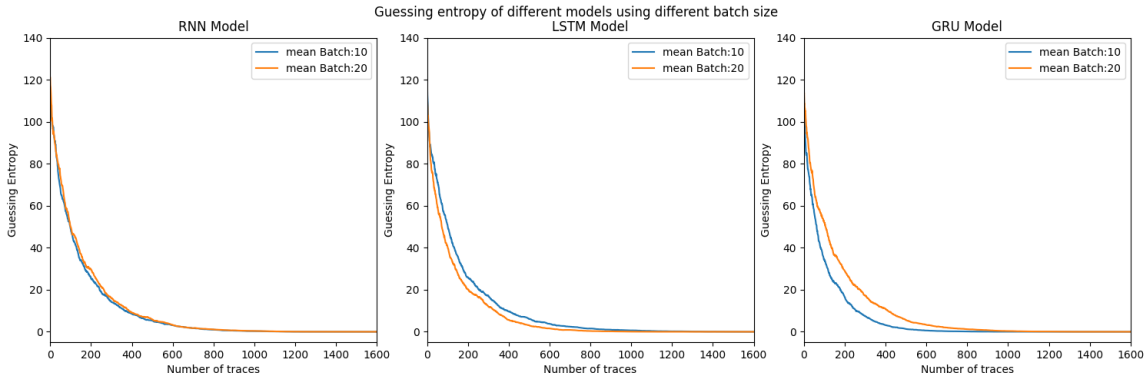


Figure 4.5: On the X-axis, the amount of traces needed to reach a guessing entropy of 0. On the Y-axis, the guessing entropy value corresponding to the amount of traces. In every picture is the other model used; the used model is written above every subplot. The plots show different hyperparameters used considering the batch size.

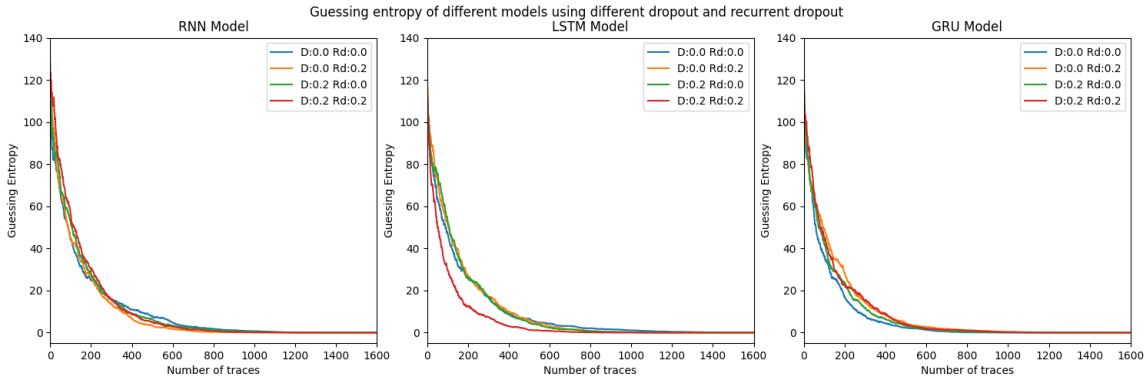


Figure 4.6: On the X-axis, the amount of traces needed to reach a guessing entropy of 0. On the Y-axis, the guessing entropy value corresponding to the amount of traces. In every picture is the other model used; the used model is written above every subplot. The plots show different hyperparameters used considering dropout and recurrent dropout.

We can conclude from Figure 4.5 that the batch size does not influence the results much considering the RNN model. Also, dropout and recurrent dropout in Figure 4.6 do not influence the model RNN model. It looks the RNN model is robust against changes in the hyperparameters. Another thing could be the vanishing gradient problem, which applies to the RNN model, or the model is overfitted. However, because we do not see this in the other two models, we assume that the size of training and testing is not too much, and therefore we conclude that the RNN is more robust to changes in the hyperparameters than the LSTM and GRU. We expect that a smaller batch size is better for the performance of the model. In the GRU model, we can see that this hypothesis is proven. For the LSTM model, we see that the difference is minimal and that the smaller batch size gave a slightly better result. However, this difference is not significant. Therefore, we can conclude that a batch size of 1/7 is right for the LSTM model and the RNN model. For the GRU model, we prefer the batch size of 1/15. Concluding the dropout and recurrent dropout value, we see that the difference is minimal between the setups. Only the dropout of 0.2 and recurrent dropout of 0.2 for the LSTM is giving a different result. No further actions have been taken from this, and the standard dropout and recurrent dropout was either 0.0 or

0.2. For the rest of this chapter, we keep using both of them to get more generic results.

4.2.3. DPAv4 Selected Time Window of Size 150

In the previous experiment, we saw a better performing sequential data model. One of the possible explanations for this could be the sequence length. Sequential data models are known to be very strong for sequential data. In other research [73], we see that sequential data models are used mostly in text and speech recognition problems. For commonly encountered problems, most sentences are not longer than 20 words. Then getting a dependency between more sentences is seen as a challenging problem. Therefore we reduce the sequence length even more in the following experiment. In this thesis, we have seen a sequence length of 3 000, and 450. To get a more brief overview of the possibilities of the sequence length that could work well, we do a third experiment with 150 values. For this experiment, all the parameters were the same as previously discussed. Meaning the recurrent dropout was 0.0, dropout of 0.0 and 0.2, batch size of 1/30, and 1/15 using one layer and the three different models. The ten experiments per setup and for every four setups, we took the total mean to represent the model and his layer for performing in side-channel data. In Figure 4.7, there is an overview of different sequence lengths with different layers and using the RNN, LSTM, and GRU.

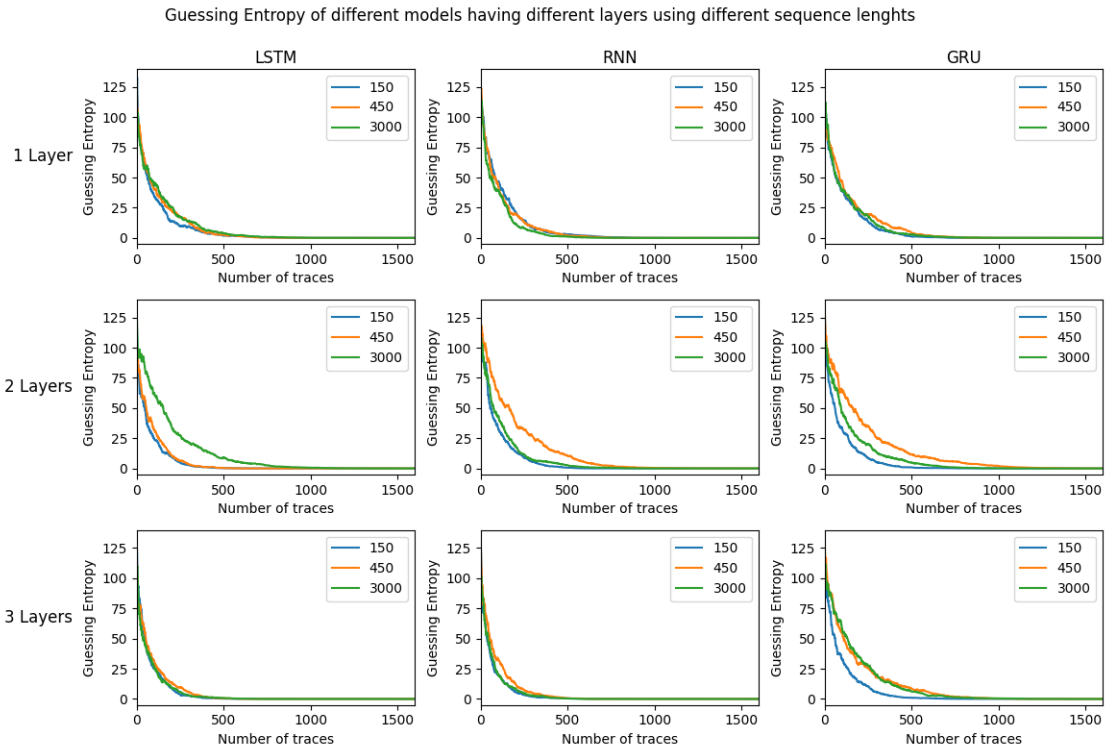


Figure 4.7: The columns represent the different models and the rows show the number of layers. In every plot are the three different sequences length. The plot is showing the Guessing entropy for the DPAv4 dataset.

All the results of the experiment can be seen in Figure 4.7. The first thing we can conclude from this experiment is that the GRU model is performing worse compared to LSTM and RNN. This could be explained because the GRU model can remember specific events for a long time, but not necessarily more. If the leakage in the trace is divided over more timesteps, it would mean that it will only remember a few leakage timesteps, which will not highly enough correlate with the key because of all the other noise there is in the trace. Moreover, we can see that decreasing the sequence length does not influence the model behavior a lot. We see the most difference in performance using two layers on all the three models and using the GRU model with three layers. However, on the most performance-wise of the amount of traces, we do not see a decrease in performance. From this, we can conclude that it does not matter for the model's performance if we use a sequence length of 150 or 3 000. This

also proves that the windows used do hold the right leakage. However, it should be considered that decreasing the sequence length 20 times does give a considerable speedup (around ten times) for the network that needs to be trained. The figure has some strange results which are not consistent. To make sure it was not a bad run of the experiment, some experiments have been rerun to be sure. However, no changes have been found.

4.3. Reducing the Sequence Length

In this thesis, we have seen that reducing the sequence length did improve the speedup of the model used. We also saw that using 150 as sequence length made the model more consistent than with longer sequences and gave the model a considerable speedup. However, using these smaller datasets was found by other papers [65]. The right window on the sequence is found by first learning a CNN on the dataset and then look at what part of the data the CNN is learning on. It is not very convenient when we put this situation to a real-life side-channel attack. Then we first have to learn a CNN (which is at the moment working better on SCA) to analyze what part of the data it is learning on and then use that data to find a window to train another sequential data model and show that the sequential data model works. Therefore, in this section, we use some preprocessing techniques on the complete sequence length to reduce the length of the sequence to speed up the algorithm's learning time, but we do not know what kind of information is hidden at which time step. First, we use a dataset with 50 features, which is already known in the side-channel community. Then we use a second approach for something entirely new for side-channel datasets. We made a sequence length reduction by using linear regression.

4.3.1. Pearson Correlation Dataset

The first experiment that was conducted is on the Pearson correlation dataset. The dataset is from [54], and there we can find the specifics about the creation. In short, the dataset exists out of 50 features per trace, which are generated with a Pearson correlation. Therefore this dataset is commonly referred to as the '50 features dataset'. The 50 features are then sorted based on the highest correlation. These 50 features dataset has a more common length from the perspective of the use of sequential data models. A trace now looks like a few sentences. The same hyperparameters as before are used for comparability. However, we now only use three layers because we saw the most promising results with that many layers before.

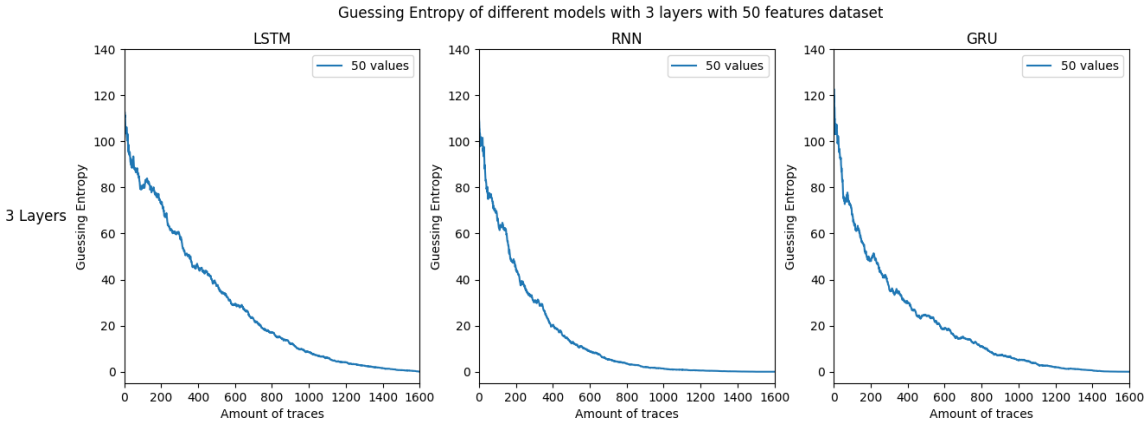


Figure 4.8: Results of three different experiments when using three different sequential data models. The dataset used in this dataset is the 50 features dataset.

In Figure 4.8 are the results of the experiment. Having a first glimpse of the guessing entropy, we can see that the results are not as good as before. Reaching a guessing entropy of zero after around 1 000 traces in the best case, which is RNN. Moreover, reaching a guessing entropy of zero after around 1600 or even more traces with LSTM and GRU. This means we can conclude that this dataset is not suitable for the sequential data model. However, what should be considered is that the dataset is not sequential anymore. Because the timesteps are sorted on the highest correlation, it is missing any form of sequential dependency, something we should have for sequential data models.

The model hyperparameters used have been concluded as the best setup for the experiment, and therefore no further tweaking should be necessary. Therefore, we conclude that this experiment has these disappointing results because of the sequential dependency that is missing inside the dataset. We need a feature reduction where the sequential dependency is taken into account.

4.3.2. Preprocessing with Linear Regression

Another smart way to reduce the length of a sequence is to use linear regression, as described in [69]. What should be remarked is that using linear regression as a preprocessing technique has not been done before in a side-channel context. In this experiment, we entirely follow the methodology, as described by the paper. We are first using linear regression to reduce the sequence length from 3 000 to 150. Using a window size of 20 and calculating a linear (Y=ax+b) formula for a and b to represents these batches of size 20. This a represents the batch slope, being positive, meaning a rising value and negative is descending. Nevertheless, the value of a represents how steep this slope is. The b represents how high the values are, the starting value of the linear line. We create two new datasets from all these 150 formulas—the first one containing all the a values per batch, having 150 values per trace. The second dataset contains the mean of the Y value per window size. This means that for the formula that is calculated with linear regression. X was all the values between 0 and 19, which results in 20 different y values. Then the mean was taken from these 20 y values. This means that the dataset with 3 000 timesteps per trace is now replaced by two datasets with 150 timesteps per trace. 150 is chosen because we had seen before that the model was operating very consistent on a dataset with that sequence length. We also keep the sequential dependencies between the data because the window that creates the batch moves from left to right. To prove the linear regression technique's correctness for reducing the model, we have an example of a trace in Figure 4.9.

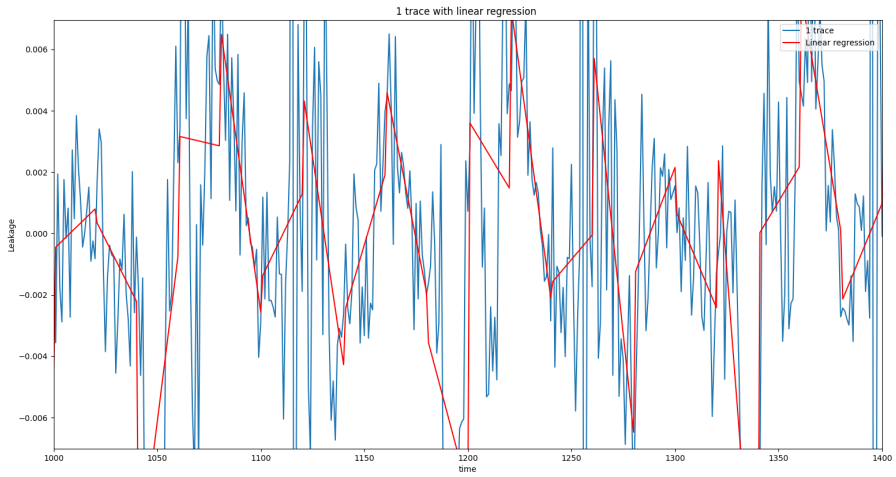


Figure 4.9: The blue line representing the original trace between timestep 1 000 and 1400. The red line shows how the linear regression fits on each window of size 20. On the X-axis the timesteps and on the Y-axis the leakage of the trace.

In Figure 4.9 we see on the x-axis the time steps and on the y-axis the corresponding leakage. With a blue line is the original trace represented. The red line shows the generated linear line for that window size. What should be noted is that the red line makes big jumps. That is happening when the window size is equal to 20. Meaning a new linear regression model starts a fit on the new batch of length 20. This jump is at one timestep and therefore does not influence the dataset. However, what could be acclaimed is that the bigger the jump, the more difference there is between the different windows. However, we see that the linear regression model is an excellent fit for the original blue line. Much noise and scatter are removed and replaced by a fluent line that does show the average regression of the original trace.

The following experiment is a replica of the previously named paper. This means we train two

separate sequential data models. In the paper, this is an LSTM, but here we also use an RNN because this sequential data model also showed promising results in the previous subsections; the amount of layers used in this paper is equal to 4 layers. The first sequential data model is trained on the dataset containing all the a values of the linear regression formula. The second model is trained on the second dataset with the mean of the Y values. This means one model keeps track of the slopes while the other model only keeps track of the heights. In the end, the output of both models is concatenated, resulting in one output, which is classified using the softmax activation function. The results of the experiment can be seen in Figure 4.10.

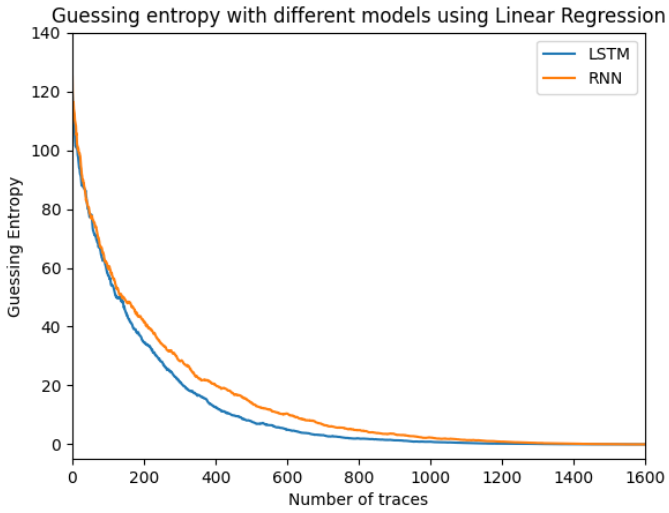


Figure 4.10: The blue showing the mean of the experiment using an LSTM model, where the orange line shows the mean of using an RNN model. Two models are trained on two different datasets, which are generated by using linear regression. In the end, the output is concatenated.

In Figure 4.10 are the results of the experiment. The same hyperparameters are used as told before with ten different runs. In the cited paper, they do not use recurrent dropouts. The mean is taken from these four different setups to summarize a model's performance with a linear regression dataset. What can be seen is that the LSTM model is performing better than the RNN. This was already seen before and only gives additional evidence that the LSTM performs better than the RNN considering side-channel analysis traces. However, when we compare these results with the results of Figure 4.7, we see that it is performing way worse than the 150 most important time window we used there.

We can also conclude there is a drop in performing compared to the 3 000 values used before from the DPAv4 dataset. We can, therefore, conclude that using linear regression as a preprocessing technique for side-channel does work. It reduces the sequence length a lot but does not improve the results of the model. However, when we compare it to more practical aspects, there is some difference. First of all, the linear regression speed is around the same as with the dataset, where we used a window of 150 values. This is because the sequence length is the same. In the linear regression approach, we have to train two separate models, but this can happen in parallel because we have two different datasets. This means we do need double the hardware to get the same speed, but the speed is around the same. Also, we use the complete trace as a starting point, which is still a bit doubtful in the previous scenario, where a window is used. Therefore the experiment is more representing a real-life scenario but does not improve the quality of the attack. Investigating what happened per hyperparameter, we can look at Figure 4.11.

In Figure 4.11, we see not much difference considering the LSTM and the hyperparameters used. The only difference is that the red and green lines for the LSTM model reaching a guessing entropy of zero a bit earlier than the other two. From this observation, there is a small suspicion that using recurrent dropout improves the sequential data model. Moreover, when we look at the RNN, we see the same improvement considering the green line (using a recurrent dropout value). However, looking at the red line, which is also using dropout value, we see that the results are terrible. We cannot

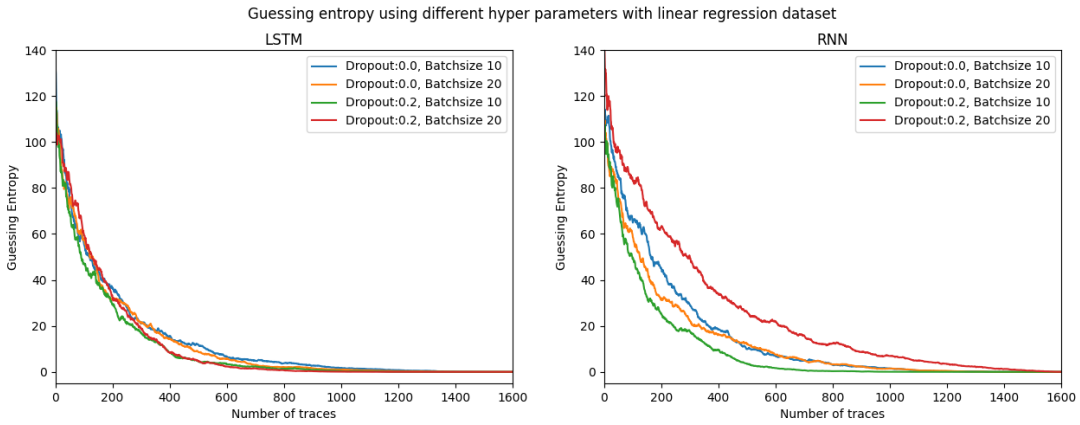


Figure 4.11: On the left, the results of the LSTM model having four layers. On the right side, an RNN model having four layers. Both plots show different hyperparameters, which are explained in the legend of the figure. The models are trained on the dataset generated by using the linear regression technique.

conclude from these fluctuating results if using recurrent dropout better to use or not in sequential data models considering side-channel analysis. To conclude, we see that the LSTM is more stable than the RNN using different hyperparameters. This could be explained due to the vanishing gradient problem that applies to the RNN but not for the LSTM. We see when we use this dataset that the RNN is more sensitive to different hyperparameters than before with the original dataset. Another reason could be that we make (timewise) jumps of 20 in this setting. If we then also drop values, the recurrence dependency is decreasing in the dataset. The LSTM shows more stability on these different hyperparameters as their results are better than the RNN. The difference of the LSTM model could explain this compared to the RNN model.

4.4. Bidirectional Layer

Until now, we have seen the sequential being able to find the correct intermediate value. However, we have not seen the sequential data models beating state-of-the-art CNN, where DPAv4 is broken within ten traces [38]. A reason for this to happen could be lying in the way a sequential data model works. The sequential data model has been explained in the previous chapter and showed their strength in understanding and reading sentences. Considering side-channel data, this could mean leakage of the side-channel trace is somewhere at the beginning, for example, in the first 100 time steps. The sequential data model should then remember this leakage when seeing another 2900 timesteps, after which it makes its classification. The first solution to fix this problem is to run the network using bidirectional layers. In the following experiment, we did run the same hyperparameters as used in the previous experiment. The only addition is that the model used has now all bidirectional layers, which means that the model also learned the reverse order of the trace. The coincidence that the leakage is in the middle of the trace, which means the bidirectional layer does not change anything, is neglected. We use the datasets seen before with a sequence length of 150 and 450 values. The model was slower than before because the bidirectional layer learned it both ways per sequence. The results of this experiment are in Figure 4.12.

In Figure 4.12 are the results of the experiment where the model uses bidirectional layers. From this figure, we can conclude that only one layer performs better than using two layers when we use a bidirectional layer. Before this experiment, we saw that using more layers would improve the results, but we see the opposite in this bidirectional context. In the model without a bidirectional layer using the size of a window of 150, we saw slightly better results. In this experiment, most of the guessing entropy would be around 500/600, where we saw without a bidirectional layer a guessing entropy of 400/500. The reason that the bidirectional layer is decreasing the performance could be due to the following. When the leakage is at the beginning of the sequence, an LSTM would learn this. However, because of the bidirectional abilities, it also has to learn this leakage at the end. Now the leakage in the trace is less time-dependent and more spread over the sequence. This makes the leakage less unique, and therefore harder to classify for the sequential data model.

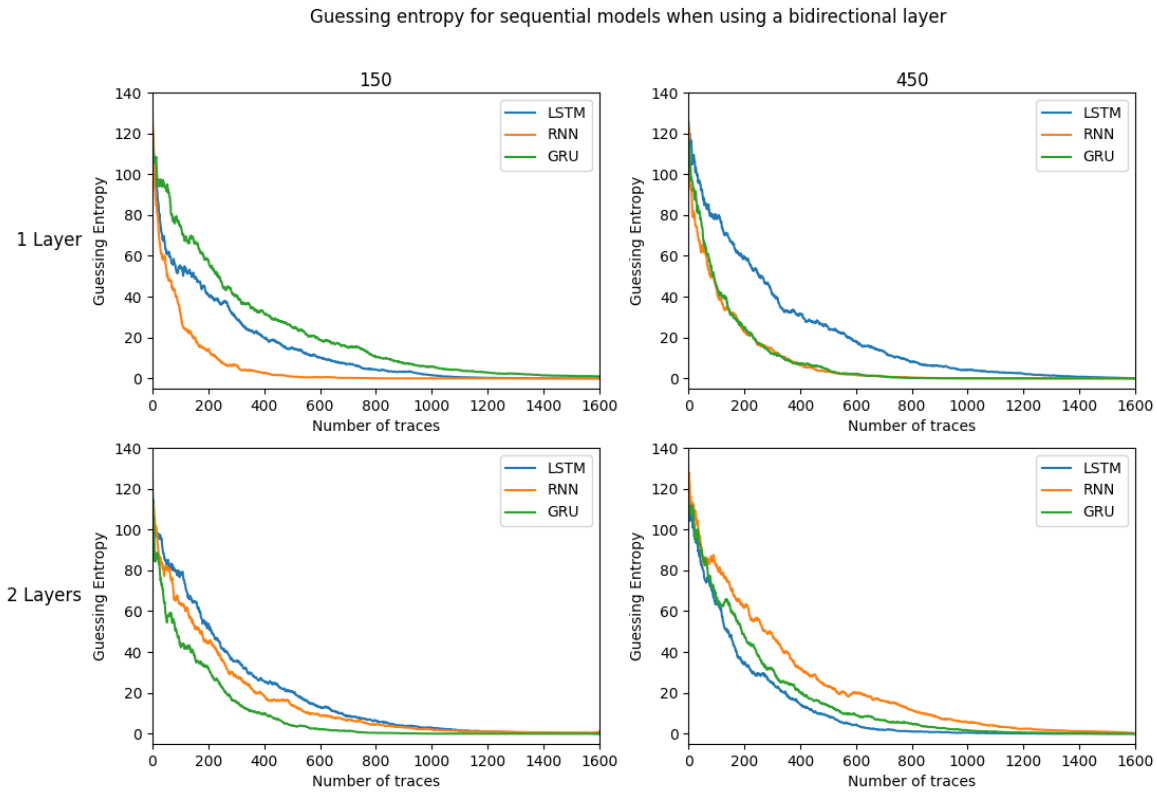


Figure 4.12: The first row shows the models using one layer, the second row are models having two layers. The first column is with dataset length 150. The second column represents experiments using a sequence length of 450.

4.5. Advice on using Sequential Data Models in SCA

Until now, we have tried a lot of different setups for sequential data models to prove they are working. Till now, we have not seen better results compared to the state-of-the-art side-channel attacks. To make the final conclusion on using sequential data models for performing side-channel attacks, we did three more experiments. In the first experiment, we used a different countermeasure. In the previous sections, we had a masking countermeasure, but now we also want a hiding countermeasure, which is the random delay countermeasure on an AES encryption. Secondly, we use a different leakage model for the DPAv4 dataset, the hamming weight leakage model [25]. Finally, we experiment with a different dataset, which is more a representing dataset in terms of countermeasures, to see how good the model performs on something that is more a realistic scenario. This dataset was the ASCAD dataset. Till now, there was much research for optimizing the sequential data model. We kept using those hyperparameters for the experiments. In this section, we did not go into the hyperparameters' optimization but are mostly interested in how the sequential data models perform over the different datasets that represent better the diversity in the side-channel domain. So we can give better advice on the usability of sequential data models in the side-channel domain.

4.5.1. AES with Random Delay

In this experiment, we used a dataset that has a random delay countermeasure. Because the DPAv4 dataset does not have this countermeasure, this could indicate an interesting dataset. Also, from the concept of side-channel analysis, it does mean we tried both types of countermeasures commonly used in datasets for side-channel analysis. The setup of the experiment and the hyperparameters is the same as before and explained in the methodology. The results can be seen in Figure 4.13.

What can be seen from both plots is that the batch size again influences the guessing entropy of the model. The left figure reaches a GE of 0 at a maximum of 500; the right figure needs at least 600

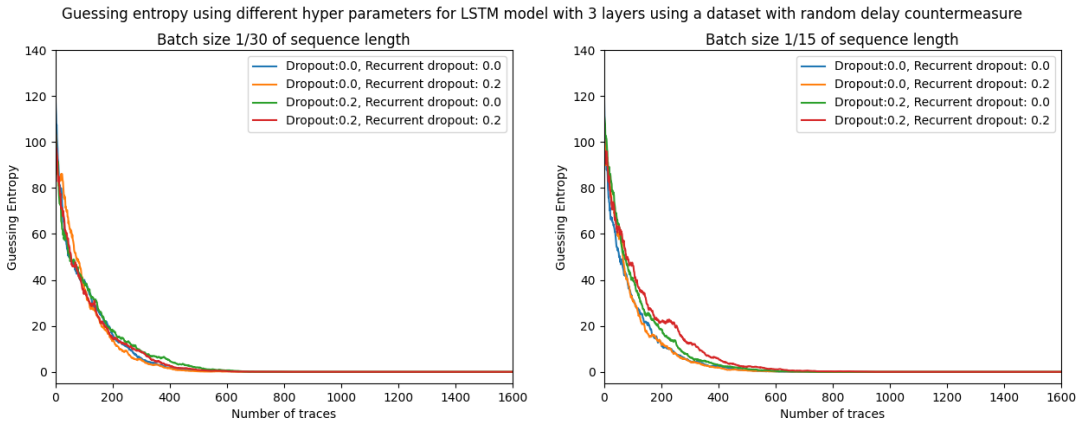


Figure 4.13: Using the LSTM model to train a dataset with random delay countermeasures. The left figure using 1/30 of batch size, and the right figure has 1/15 of batch size. The plots have Guessing entropy on the Y-axis and the number of traces on the X-axis. Furthermore are the different setups plotted in the figures.

traces. We see, however, slightly better results compared to what we saw before. The best results for the smallest batch size is around 380. Therefore, we could conclude that the sequential data model is better in dealing with a random delay countermeasure than with the masking countermeasure. Some tiny shifts relative to some point do make less sense. The same happens when we read a language as a person. Two words that are switched is sometimes something we neglect while reading. See the following example:

I got a dig bilemma
You that read wrong
you read that wrong too

Most people do read the first two lines without any problem and understand what you say. However, looking at it a second time, you see some spelling mistakes in the sentences. We expect to read something wrong and therefore interpret the sentence before understanding it entirely. It shows that relative orders do not matter. However, mixing the complete order (wrong you that read), you probably recognize it immediately. The sequential data models work in the same way and therefore do work better than with the DPAv4 dataset.

4.5.2. Hamming Weight Leakage Model

Now we have tried a different countermeasure. We also want to experiment with a different leakage model. Therefore we did experiment in this thesis with the hamming weight leakage model. The dataset is commonly used in the side-channel domain [25]. There could be some more exciting results in this experiment because we now use many to one classification. This does not happen a lot in the sequential data model. Moreover, when it is used, the amount of classes to classify to is ideal between two or four; until now, we have seen 256 classes. Therefore the hamming weight dataset has some potentials and should be experimented with to conclude if sequential data models are suitable for side-channel data. This experiment did tell if fewer classes for classification is better for sequential data models. For this experiment, we did only run the best performing models we have seen. That means a model with three layers and taking the mean of the setup of the different experiments.

In Figure 4.14 are the results of the experiment. The difference in each model's performance is less than seen before, which could mean that the different networks are more consistent. The difference in how the models are performing is more the same then seen before. This could be because the possible classes are between 0 and 8, and therefore, the difference in relative values is also less. The spread of values is more central to the value of 4. Looking at the results in Figure 4.14, the amount of traces for reaching a Guessing Entropy value 0 is around 1600. Which means that the model is performing way worse than seen before. From this experiment, we can conclude that using the other leakage model does not improve the side-channel attack. For now, we can conclude that a different leakage model is

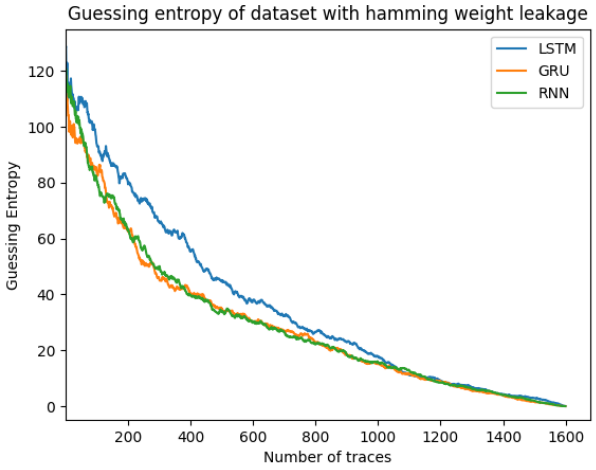


Figure 4.14: The results of the experiment with hamming weight dataset. For this experiment 3 different models are used all having 3 layers.

not changing the results we have seen before. Therefore we have additional evidence that sequential data models should not be used for classification tasks in the side-channel analysis.

4.5.3. ASCAD Dataset

We have seen before that using a different leakage model does not improve or change the results we have till now. The baseline we created is needing around 300 traces to reach a guessing entropy of zero. Till now, there is a great preference toward not using a sequential data model to classify side-channel data with the right intermediate value. All the experiments seen before this chapter are done using the DPAv4 model and the value leakage model. For this experiment, we use our best performing model to have a guessing entropy of around 300 to attack the ASCAD dataset with the fixed key. More information about the ASCAD dataset can be found here [56]. We did use the best performing setup. Which is using three layers with a sequential data model without a bidirectional layer. The results of this experiment can be found in the following figure. We did attack the third key byte value, which is masked.

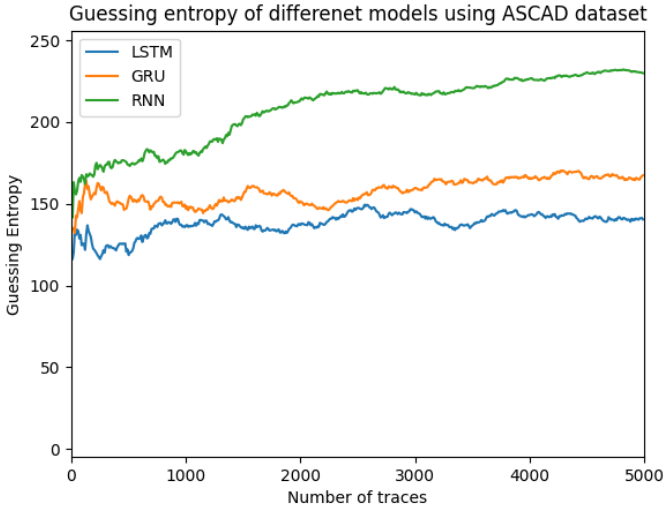


Figure 4.15: The results of the experiment with the ASCAD dataset. For this experiment, we used three different models, where every model has three layers.

In Figure 4.15 are all the results of the previously explained experiment. What can be seen is that none of the models can predict the right intermediate value. We used the best performing model setup with the DPAv4 dataset. Therefore we can conclude that the used sequential data models are not able to predict the ASCAD dataset. It even looks like the models are learning the same intermediate value, and therefore we could think that all the models are overfitted on a specific intermediate value or that it was learning something but not the right thing. From this experiment, we can conclude that using a different (and harder) dataset does not work with the sequential data model. We did not investigate the use of sequential data models any further for the ASCAD dataset because none of the models can converge to the right intermediate value, and considering the best performing setup, we feel this is at the moment the optimum for the sequential data models.

4.6. Conclusion

This chapter has a broad exploration of the RNN, LSTM, and GRU for side-channel data. The chapter aims to explore how the previously named sequential data models can be used and find out which hyperparameters should be used for the sequential data models. Furthermore, a recommendation on if and then how to see the further use of sequential data models in the side-channel domain.

In this chapter, we have experimented with different hyperparameters when using side-channel data. We have compared different batch sizes, units, dropout, recurrent dropout, activation function, training size, and layers for the RNN, LSTM, and GRU model. We even tried using a bidirectional setup for sequential data models. We found that using recurrent dropout is mostly decreasing the model's speed and does not improve the convergence a lot. Also, increasing the training size did not improve the results. We came to the conclusion that we should use the He Uniform weight initializer and that the sigmoid activation function is working better than the TanH activation function.

We saw a significant improvement when the sequence length was reduced, which is concluded from most sequential data models research. Where they show that an LSTM does not work with a sequence length of more than 1 000 [73], sequential data models follow the classic 'Less is More' statement. We used a new preprocessing technique with linear regression, and with that, we were able to reduce the complete sequence length to something that the models were able to learn. After that, the model was performing the same and therefore proved that using linear regression as a preprocessing technique is a good alternative for the large sequences in side-channel data.

Furthermore, we concluded that the number of units should be equal to the dimension, which was one in this case. For batch size, we recommend a small value, which, of course, decreases the run time. Recurrent dropout is best lowered to 0, which gives the model GPU speedup. Use three layers where possible, and preferable the LSTM model.

In all the experiments, we have seen the best of needing 300 traces before having the correct intermediate value at key guess position 0. There are two ways to compare these results. Suppose we compare it with the state-of-the-art sequential data models in side-channel analysis [25], where they have a guessing entropy of 1 000. We could say that this research made a significant improvement in using sequential data models in side-channel. Comparing these results with state of the art for side-channel with DPAv4 dataset using any model possible, the results are relatively worse. There they can find the intermediate key-value after having ten traces. Therefore, we conclude that the sequential data models do not have comparable results considering side-channel analysis. Moreover, we want to address that sequential data models are not advised in the many to one classification where we try to classify the trace with all the time steps to an intermediate value.

First of all, there is an intuition that the traces have not enough leakage to classify one intermediate value. What can be seen from this chapter is that the sequence length influences the performance. Having a leakage at 20 timesteps at the beginning of the trace and then remember it to the end where it classifies seems impossible for the sequential data model. If we compare this to a book with one sentence that holds the essential information of the whole book, but the rest of the story is just jitter and noise. Then it is hard for an algorithm to learn to recognize that one sentence. For the trace to be classified the right way, we should need a high signal to noise ratio over the whole trace. Even more, they showed not to be sensitive for this data when using bidirectional layers—therefore concluding that it was less critical where the leakage was in the trace. However, the model was not good at capturing it.

Furthermore, we see in the last section of this chapter how the sequential data model responds to

different datasets. We see a slight improvement when a dataset is chosen with random delay countermeasures. We, therefore, get the assumption that the sequential data model is performing better against hiding countermeasures, mostly random delay, then the masking countermeasures, which does change the leaked signal. Looking at the hamming weight leakage dataset, we see a decrease in performance. The same happens when we use another dataset, which is more common in the side-channel analysis. If people want to keep using this, more research should be done to find a way how it will give more comparable results; however, in this research, we have not found this. Things that should be considered when working further in these models are summed up. The hardware issue a sequential data model gives, but even worse, the time complexity. It also does not help that GPU speed up is unavailable when we use recurrent dropout. Maybe even more vital is the example of reading a sentence, that we do not look at the relative position of a word in a sentence, but we look at the big picture and take some critical words out of it, which is more the case in a CNN. Lastly, we want to address that sequential data models show to be a robust model considering text. Here every word in a sentence has a meaning; in other words, there is almost no noise in the sentence. Comparing this with side-channel analysis, we see a big difference. Here the traces do have much noise; the datasets are not comparable.

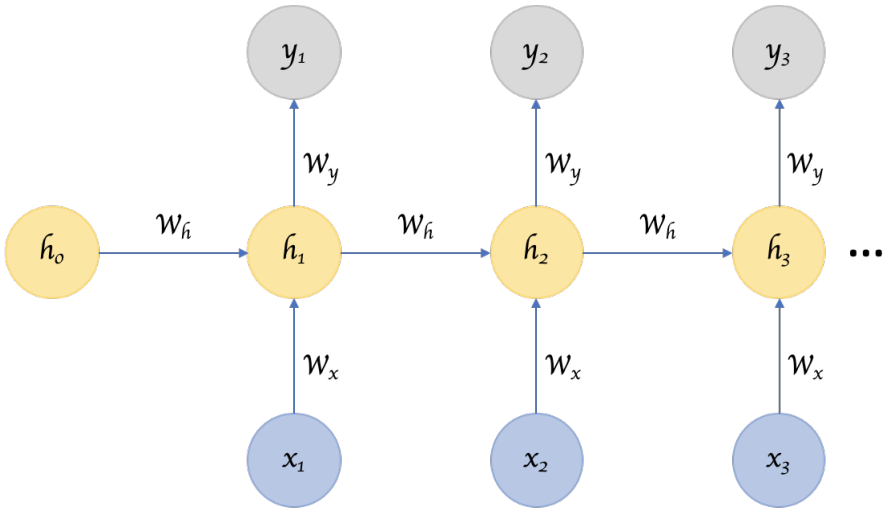


Figure 4.16: A visualization of a sequential data model. Where y_n is the output of a cell at a specific time step n, x_n the input at a specific timestep x and w_h the weight that is given to the next time step.

A more theoretical explanation is because the problem lies in the way the sequential data model is built. In Figure 4.16 there is a visualization of an sequential. The problem of a sequential data model lies in this visualization. This image counts the same amount of cells a sequence length, meaning in our case, 150, 450, or even 3 000. The y_n where n is equal to the sequence length is used for classification. However, this does mean that the vector y is holding all the information of every step before. This is a nice opportunity to make mistakes, and therefore, we see sequential data models, not as an option. We also see that big companies are starting to drop these networks and start using attention instead.

In our opinion, a sequential data model should be used in many to many classifications. Where we see, it is used now by Siri from Apple, Google assistant from Google, etc. In this standard classification problem, as described in this chapter, this is a challenging task. That is because there is no dependency between the output of the intermediate values. That means that when the correct intermediate value is 108, but the algorithm predicts 90, it is as wrong as when it predicted 11. There is not something like being close and therefore performing better in side-channel. There should be some dependency between the input and the output of what is being classified for this to work if there will be more research in the many to one classification with sequential data models. We advise the usage of attention.

5

Denoising with Autoencoder

As discussed in the previous chapter, the sequential data models have not been shown to outperform the state-of-the-art side-channel attacks. However, it was opted to use it for many to many classifications. This means the data holds a dependency between the input and the output. In this chapter, we explored this even further. Sequential data models are used in the natural language processing domain; they need to translate a sentence written in language A to a sentence written in language B. We are going to use this same idea to remove noise from a trace. In this chapter, we first dive into the theoretical knowledge about translating this text to text problem to a trace to trace the problem. In the second section, we used an autoencoder to translate the noisy traces to a clean trace. In the third section, we set a baseline and show how a CNN model is influenced when learning on clean traces and attacking noisy traces. The fourth section is an in-depth analysis of how our autoencoder is built. In the fifth section, we show the CNN model's performance on the newly generated clean traces. The last section is a conclusion of the complete chapter.

The setup for this experiment is of great value for the side-channel community. This is because we see many more countermeasures taken place. These countermeasures should make it harder for the attacker to use the traces produced by the hardware. The research at the moment is mostly interested in making neural networks better to deal with countermeasures. However, the contribution to removing the countermeasures before we use a model to learn is neglected. Furthermore, this research could be the first step towards a unified cleaning process. Imagine a cloud where we can send the original noisy traces, and these traces are then cleaned and returned such that a model is better in attacking it. If this is possible, we should know how good this works to make better countermeasures. Therefore this chapter is of great value. Finally, while capturing a trace, there could be inconsistent results. This translation solution could be able to correct this problem.

5.1. Translation Problem

Translating a sentence from language A to B is one of the most challenging tasks. A language is a complex mechanism with specific grammar, own verbs, and every language is using their way of ordering words. Even then, there is some way of interpreting a sentence and its context. When the sentences belong together, the model should remember part of the first sentence and use that for the second sentence. For example, a king talks to his people when we want to say in the second sentence that he was talking loud. We should know that it was about a king and that a king is a male and therefore 'he' should be used. All these previous problems make it quite complex to translate one sentence to another. Because a sentence is a sequence, this problem is mostly referred to as a sequence to sequence problem.

When we consider a sequence to sequence problem (seq2seq), the most used model is an autoencoder. Looking at the problem described in this chapter, we could learn the autoencoder on noisy data as input and original data as output. After that, we could clean never seen noisy data to clean data and perform a more efficient attack. We did not evaluate the translated traces with sequential data models. We have seen in the previous chapter that the sequential data models are not sensitive to hyperparameter tuning. Furthermore, we know that the models do not give a state-of-the-art perfor-

mance. Therefore we evaluated the model with a CNN. Even more important is that there is already some research about cleaning noisy data, which is also evaluated with a CNN. Therefore evaluating it in this thesis with a CNN does make the results more comparable.

5.2. Methodology

In the following sections, we experimented with the proposed translation model. However, first, we describe the methodology used to evaluate the proposed experiment.

We experimented with a sequential data model to ”translate” a noisy trace to a clean trace. Moreover, we did like to know if the model is performing as good on the cleaned trace as on the original trace. We evaluated this by using the most common dataset in side-channel analysis, the ASCAD dataset. The specifics of this dataset are discussed in chapter 2. ASCAD is used because other preprocessing cleaning algorithms use this dataset as well; therefore, we can compare our results. Our problem classification was sequence-to-sequence; here, we need clean and noisy data. To get this data, we artificially created some noise in the dataset. For clarity, the traces obtained that are in the original dataset are called the original traces. We artificially add noise to these original traces, which generated a dataset called noisy traces. We learned an autoencoder on these two datasets and then made predictions on the noisy traces; the results are called the clean traces. We used three different noise generation techniques to show the result of our autoencoder. The following hiding countermeasures are used to generate noisy data:

- Gaussian noise countermeasure, here we add a uniform value between -20 and 20 to each point of the trace. In Figure 5.1, we see the result of the Gaussian noise that has been added to the original trace.
- Desynchronization countermeasure, with a maximum of 50 data points. At every trace, a random uniform integer is generated. This integer is equal to the amount of desynchronization noise added (how far it looks in the future), and therefore the number of steps this desynchronization was added. The desynchronization noise is thus equal in once trace but differs per trace. In Figure 5.2, a original and noisy trace is visualized.
- Random delay interrupts, based on the floating mean method. The trace is fully scanned, and random delay is added with a change of 50 percent at every point. In case this is added, an interrupt is added with a value of 10, which was added a random amount of times. An example of a trace can be seen in Figure 5.3. What can be seen is once the interrupt is added, a recurrent sinus is visible in the trace.

The autoencoder was then trained with noisy data and the original data. After that, the model did only get noisy data as input, and then the model did generate clean data. This new clean data (which is different from the original data) is then used with the best performing literature model, a so-called baseline model. Generating the guessing entropy of our clean data and comparing the graph with the guessing entropy of the original data, we can conclude how effective our model has translated it. This methodology and noise generation techniques are comparable with the convolutional autoencoder used in [72]. We used only one type of autoencoder, which was built with LSTMs.

5.3. CNN Baseline

First, we need to have a CNN baseline to have an overview of the current reactions of the CNN best model. When the model is trained and evaluated with the original ASCAD traces, it can reach a guessing entropy of 0 after around 100 traces. Furthermore, we want to show the effects of the three noises in the traces. Therefore we train and evaluate the model on the noisy datasets. The results of a CNN trained and evaluated with desynchronization traces are in Figure 5.4. Here we see the model can converge to a key, but this is not the right key. This is not surprising because a CNN is good at handling desynchronization, but it converges to the wrong key because of the switch in timesteps. What should be considered is that the hyperparameters are not tweaked. This experiment is also proof that something has to be done with the countermeasure because CNN can remove the countermeasure without any preprocessing.

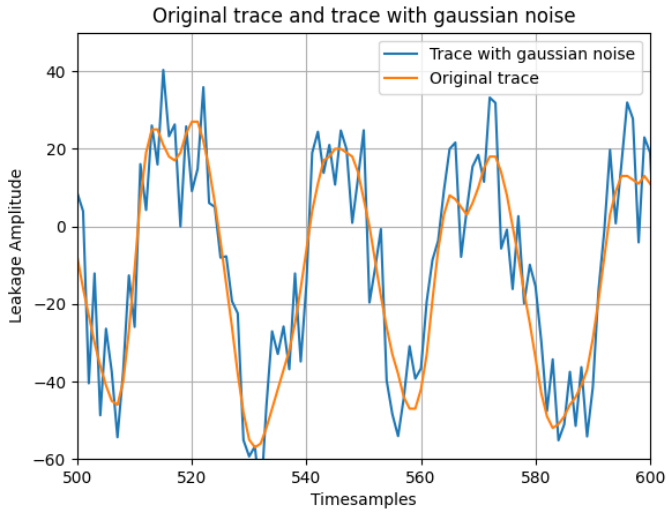


Figure 5.1: A visualization of the artificially added Gaussian noise. Where the blue line represents the trace with noise, and the orange line is the original trace. What can be seen clearly from this figure is, for example, the leakage between 540 and 550. The orange lines show a nice fluent line. However, the trace with noise looks much more unstable because of the noise that is added. Just before timestep 520, we see two nice spikes up and down in the trace with noise. Here we see that the minimum and maximum noise that is added is between 20 and - 20.

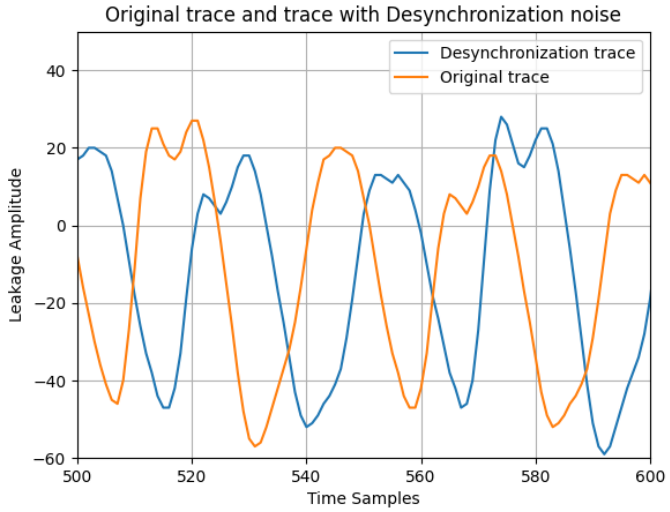


Figure 5.2: A visualization of the artificially added desynchronization noise. Where the blue line represents the desynchronized trace, and the orange line is the original trace. What can be seen clearly from this figure is the peak around 520 from the orange line is shifted around 50 places to the right and therefore identical in the blue line around 580.

In Figure 5.5, we see the model which is trained and evaluated on the traces with Gaussian noise. Here the model is not able to converge anymore. The reason is that the model should again be tweaked to the new traces that vary much more than before. These two experiments also indicate that the desynchronized traces are more accessible for CNN to break the traces than with Gaussian noise. This can be explained because the signal to noise ratio is barely different when using only desynchronization countermeasure. The leakage present in a trace is identical but only at another timestep. Changing this in the whole dataset with a random desynchronization value resulted in some different situations. However, because CNN is trained and attacked on a dataset with desynchronization, the differences

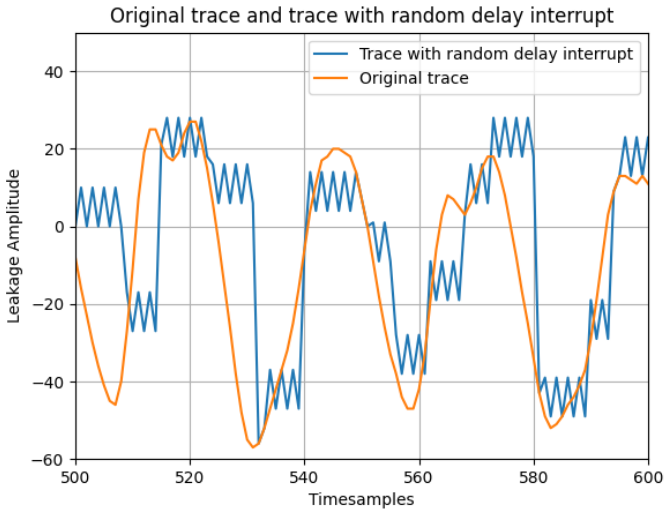


Figure 5.3: A visualization of the artificially added random delay interrupt noise. The blue line represents the random delay interrupt trace, and the orange line is the original trace. The noise is visible by the spikes that are present in the noisy dataset.

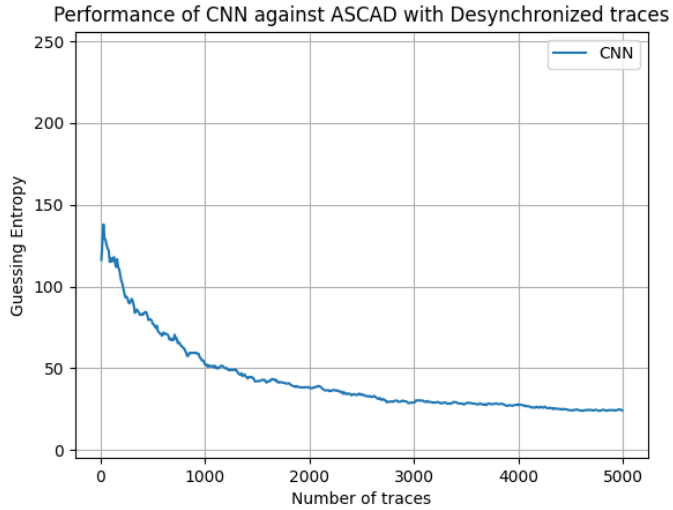


Figure 5.4: A CNN trained and evaluated on traces with a desynchronization countermeasure. What can be seen from the plot is that the CNN is able to converge but the correct intermediate value is not the most likely key.

are minimal. When considering the Gaussian noise, the signal to noise ratio is changed a lot. This explains why the guessing entropy for ASCAD with Gaussian noise is performing worse.

In Figure 5.6, we see the model trained and evaluated on the traces with random delay interrupts. Also, here the results are not better than before. This is because the leakage of the traces can be tweaked and therefore removed from the trace. For desynchronization, the noise was still present, but at another timestep. However, in this example, the noise is removed and, therefore, more challenging for a CNN to attack. Also, extra noise is added at random points, which can be seen because the nice fluent line ASCAD has is removed. This countermeasure does alter the original trace a lot. Therefore we expect this countermeasure to be the hardest for the LSTM autoencoder to break.

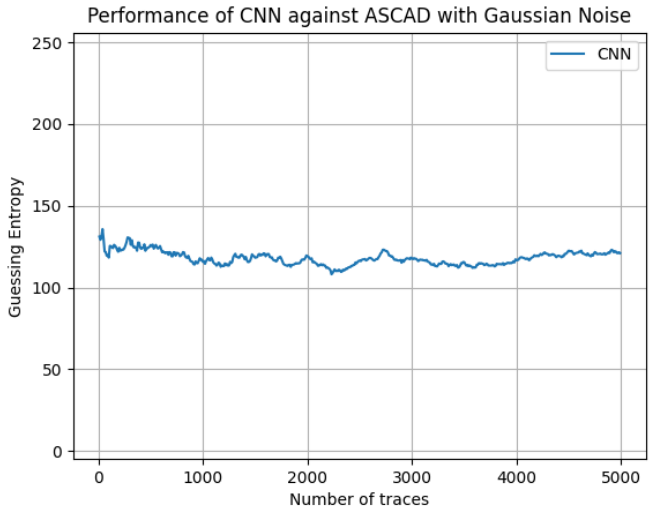


Figure 5.5: A CNN trained and evaluated on traces with a Gaussian noise countermeasure. What can be seen from the plot is that CNN is not able to converge. This can be explained because the Gaussian noise is added to every point in the trace, meaning the signal to noise ratio is changed a lot.

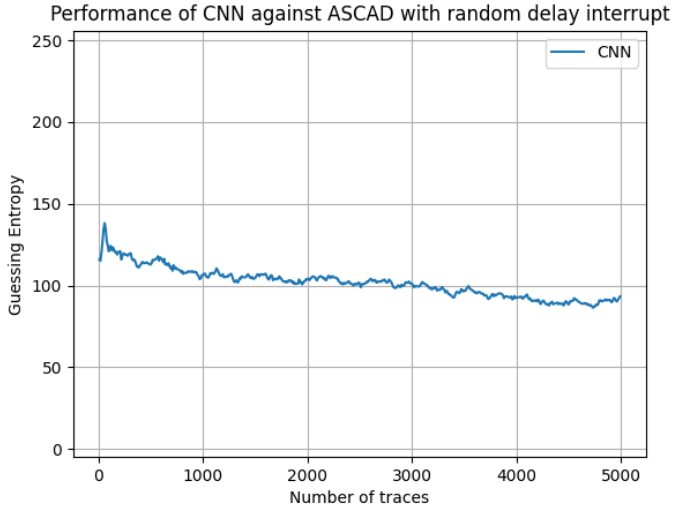


Figure 5.6: A CNN trained and evaluated on traces with a random delay interrupt countermeasure. What can be seen from the plot is that CNN is not able to converge. This can be explained because the random delay interrupts noise changes the signal to noise ratio a lot by adding the spikes at random places in the trace.

5.4. Autoencoder

In this section, we discuss how the autoencoder is built to clean the traces. This cleaning algorithm is a simple autoencoder build from LSTMs. This means the first LSTM takes a sequence input and encodes this input. After that is an LSTM that decodes the output and makes a prediction using a dense layer. A challenging step for the LSTM autoencoder is the difference between sequence length. If we compare this problem to translation problems, it makes sense that a sentence in one language has a different number of words than a sentence in another language. However, in our use-case, the 'sentences' are the same length, which means we do not have this problem. Between the two LSTM blocks of encoders and decoders is a repeat vector layer. This layer repeats the encoder's output an N times before the encoder's output is passed to the decoder. Using this repeat vector layer, the encoder's output is passed N times to the decoder, which results in time distributed output instead of an output

of time one. The autoencoder has eight layers, 4 LSTM encoder layers, and 4 LSTM decoder layers. The last layer is a time-distributed dense layer. The input is scaled with a min-max scaler before the autoencoder is used. The amount of units used inside the autoencoder is equal to 256, 128, 64, and 32; in that order. For the decoding part, the amount of units used is the same; however, in the opposite order. There is no recurrent dropout because we want GPU speedup, and for dropout, we used 0.2. The amount of epochs is equal to 200. We have used fewer epochs and fewer layers, but then the model cannot capture the whole trace. The model is trained on 25 000 traces and then predicts the other 25 000 traces. The model is trained in around three to four days.

5.5. Results

In this section, we look at the results of the experiment. First, we have learned the models to clean the datasets. After that, we evaluate the cleaned datasets with a guessing entropy function. The models that are trained are CNN models similar to the one used as a baseline. We first take a look at the new traces. After that, we trained a network and evaluated this with the guessing entropy function. This means CNN is not tweaked for the dataset, which could mean the results could be worse than expected.

5.5.1. Comparing New Traces

The autoencoder has ”cleaned” the traces, and that has resulted in a new dataset. This subsection looks at the difference between the original, the cleaned, and the noisy traces. From this comparison, we were able to get a first glimpse of how well the trained CNNs would perform. In Figure 5.7 we see three different traces. In a time window from 500 to 600, the same time windows as used before. One trace colored in orange is the original trace; the blue trace is the original trace with noise, in this case, desynchronization noise. Then the green line is the clean trace. Thus, the autoencoder predicted this line, trying to be the best fit on the orange line. What can be seen in the figure is that the fit is good with the original trace. However, what we also see is that the cleaned trace is a smoothed version of the original trace. Some leakages are neglected, for example, the leakage between timestep 560 and 580. This could have dramatic consequences. In the next section, we attacked these traces to see what this means for the attack.

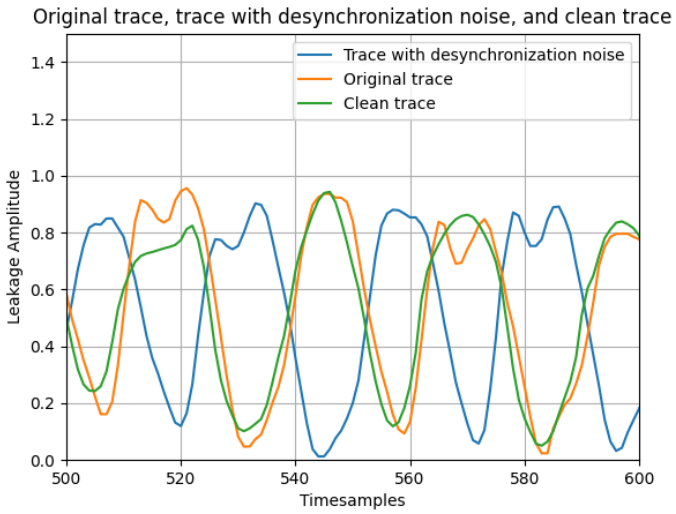


Figure 5.7: In this figure we see three different traces. The original traces, the desynchnorized traces, and the cleaned traces. The traces have been cleaned using an LSTM autoencoder.

In Figure 5.8, we see the same as the other figure but now with the Gaussian noise. The time-window is slightly different, but the colors used are the same as before. We see here that the clean trace is a perfect prediction on the original trace. So good that we barely see the original orange trace because the green line is lying on top of this one. The leakage at point 620 is slightly different from the original point but still follows the line smoothly. This figure suggests better results than the previous one

with the desynchronization countermeasure. We expect that this autoencoder's result is better because the values at specific timesteps correspond to the original dataset. We only have to remove the added noise for Gaussian noise, which can be no more than the maximum added noise (between -20 and 20). In the case of the desynchronization countermeasure, we have to shift it way more. For example, looking at the leakage just before timestep 520, there the difference is 60, and only ten steps after that, we see a difference of 0. This makes it much harder to come with an average cleaning algorithm.

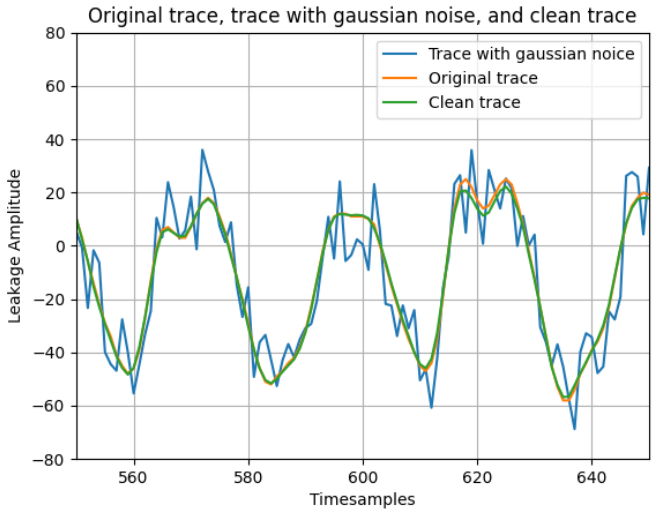


Figure 5.8: In this figure we see three different traces. The original traces, the trace with Gaussian noise, and the cleaned traces. The traces have been cleaned using an LSTM autoencoder.

Lastly, in Figure 5.9, we see three different traces but now, after using the random delay, interrupt countermeasure. We see that the cleaned traces have much more trouble to follow the original dataset nicely from this figure. At the beginning of the dataset, it finds the dataset quite nicely (see timestep 120 till 160), but after that, it gets more unstable, and around timestep 240, you see the difficulty it has with capturing the original dataset. This result could be explained with the following. At first, we think that the countermeasure used is more challenging than the two datasets seen before. The interrupt that is present is hard (harder than Gaussian noise). However, the fact of a random delay that we saw before makes the combination a tough countermeasure. We expect to capture this countermeasure still nicely if we increase the capabilities of the autoencoder. However, this means the autoencoder should increase in size, and therefore, even longer experiments have to run before capturing and nicely removing the countermeasure. In the side-channel analysis, we like to seek fast and cheap solutions. If the university's server with a lot of computational power cannot run and create a clean dataset following the original trace nicely within four days, we do not expect that other people can even use this type of attack. Furthermore, the tensors created for such a big autoencoder are too big to run on the current GPU used in the cluster. So meaning that if anyone else wants to create this autoencoder himself, this person needs costly equipment. At the same time, there are other cheaper and better solutions available. Therefore we neglect the option for increasing the autoencoder and accept how the traces look like now. We try two different datasets for the attack, the whole dataset, which is not nicely fitted at the last part of the trace. But also the trace with a window from 0 to 300. Chopping of the part of the trace, which is not nicely aligned with the original trace.

We still want to evaluate the newly generated traces. Therefore we want to evaluate the technique for side-channel analysis, and because of that, we want to evaluate it with the guessing entropy function. It is not always 1 to 1 that the results are worse if the traces are not identical. Therefore in the following section, we trained a CNN on the generated cleaned traces and evaluated these with the guessing entropy function. These results were compared with the actual results and noisy results. After that, we can say if the cleaning has helped anything.

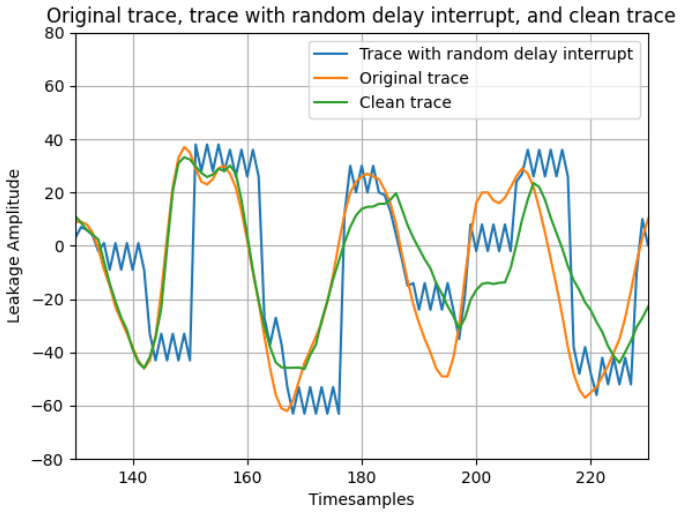


Figure 5.9: In this figure we see three different traces. The original traces, the trace with Gaussian noise, and the cleaned traces. The traces have been cleaned using an LSTM autoencoder.

5.5.2. Attack After Cleaning by Autoencoder

Now, the cleaned traces are ready; we want to know how well these perform when performing a side-channel attack. In the first experiment, we have trained a CNN with the same hyperparameters as seen before, so no further tweaking to the model has been done. In Figure 5.10 are the results of a CNN trained and evaluated on the cleaned traces. We see from this figure that the CNN cannot converge to a guessing entropy of 0. This means that for this cleaning algorithm, the experiment has failed. It also shows that if it is compared with Figure 5.4, the results have gone worse. This can be explained because the guessing entropy we see in Figure 5.4 is trained on traces where the same desynchronization is used in training and attacking. More importantly, the leakage is still present in the trace only at another timestep. In our cleaned generated traces, we see that the leakage is neglected, and mostly, the direction of the trace is followed. Thus, we can conclude that the cleaning algorithm must identify and reproduce the trace's leakage instead of following the original trace. Where the shape of the original trace was quite nicely followed, the guessing entropy has gone worse. Therefore we conclude how important it is that smoothing is not happening.

Secondly, in Figure 5.11, we see the results of a CNN trained and evaluated on cleaned traces that had been infected with a Gaussian countermeasure. From the figure, we can conclude, if we compare it with the results of 5.5, that the attack has been improved. In the guessing entropy with only noisy traces, we saw that the intermediate key-value was precisely in the middle, around place 126. In this attack on clean traces, we found the correct intermediate key-value at around 80, meaning that the intermediate key rank has been improved. Nonetheless, it is still a disliked guessing entropy plot in terms of converging. We expect that an issue could be lying in the fact of fewer traces to train on. In this example, the CNN is only trained on 20 000 traces where it is usually trained on 50 000 traces. When looking at the clean traces, we did not expect these results to be this bad. Another reason could be that CNN is not optimized for the dataset.

Lastly, in Figure 5.12, we see the results of a CNN trained and evaluated on cleaned traces that had been infected with a random delay interrupt countermeasure. From the figure, we can conclude that if we compare it with the results of Figure 5.6, the denoising made the effect even worse. We could argue from these results that the countermeasure was too hard for the autoencoder to deal with. For the guessing entropy function, we have tried both datasets. The dataset results, which had a time window between 0 and 300, are performing far worse than the dataset with the complete trace but having a bad fit after timestep 250. In the figure, we see the guessing entropy of the second named dataset, making it more comparable with the other figures. We can thus conclude from this experiment that has seen impossible for the autoencoder to denoise the random delay interrupt from the trace and make clean traces to make a useful classification.

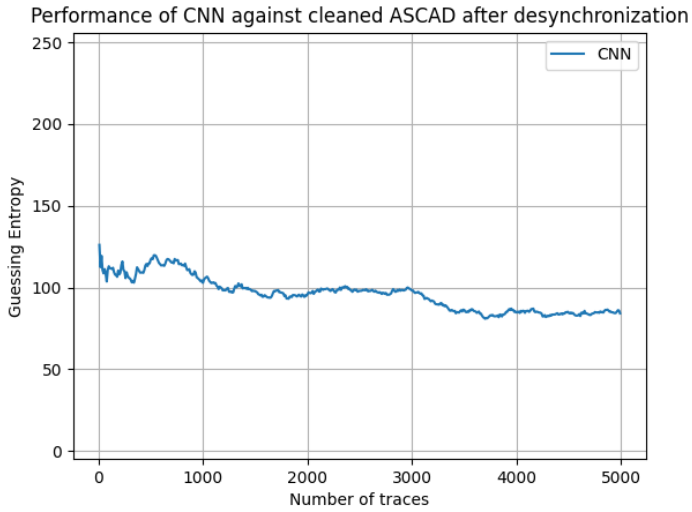


Figure 5.10: The guessing entropy plot of a CNN trained and evaluated on clean traces. The traces were dirty by adding a desynchronization countermeasure.

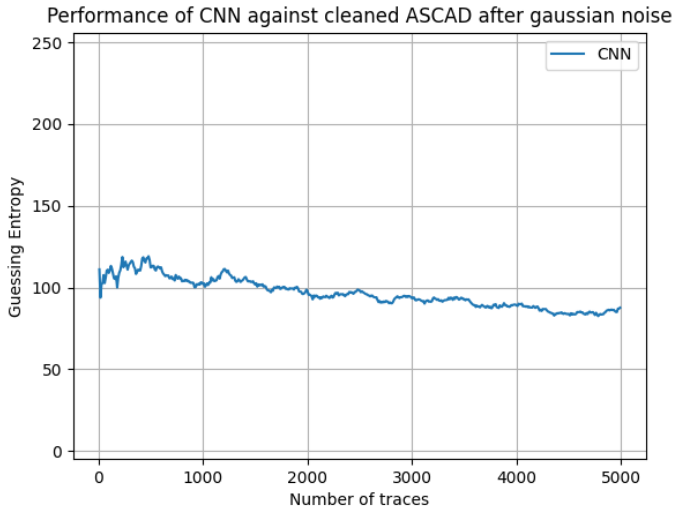


Figure 5.11: The guessing entropy plot of a CNN trained and evaluated on clean traces. The traces were dirty by adding a Gaussian countermeasure.

5.6. Conclusions

In this chapter, we have looked into the useability of using an LSTM autoencoder. We came with this solution because we saw no further advantages when using sequential data models in a many to one classification setting but preferred a many-to-many classifications. Looking at side-channel analysis, one of the use cases to do this is with generating a cleaning algorithm. An initial thing to look at was considering a technique called attention. Something used quite often in combination with sequential data models. We wanted to design an autoencoder cleaning algorithm. In the first design stage, we already found out that using attention is not necessary. Therefore, this chapter's first conclusion is that an autoencoder without attention is strong enough for many to many classifications when considering side-channel analysis traces. We can explain this conclusion with two arguments. The first argument is based on the results of random delay countermeasures and Gaussian noise countermeasures. We have seen that both countermeasures can be removed when using an autoencoder without attention. With Gaussian noise, the fit is perfect, almost identical. Considering the desynchronization, we see

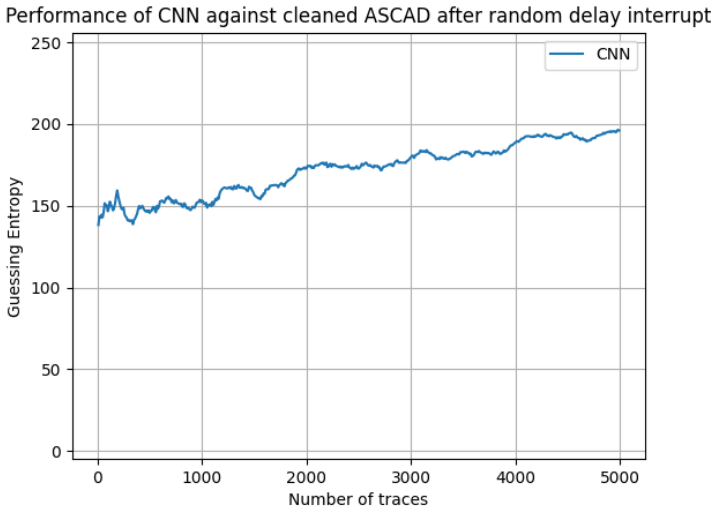


Figure 5.12: The guessing entropy plot of a CNN trained and evaluated on clean traces. The traces were dirty by adding a random delay interrupt countermeasure.

that the original trace's fit is quite good but not yet perfect. To conclude this first argument, we see that the fitting process when using an LSTM autoencoder is quite good. The second argument is time-related. Considering the results of an LSTM autoencoder without attention. We see that it already takes around 3 to 4 days before we have good results that follow the original trace. When using the attention technique, we have to do an extra calculation step per timestep, which we also have to use when generating the new dataset. This means it creates 700 extra arrays per timestep, dependent on which attention technique is used, looking at different intermediate time steps. This means this autoencoder's size with attention will increase significantly, resulting in even more time to produce the model. Trying to use this setup in the current HPC of the university, we found out that the GPU's tensors were too small to use. This means that if we want to use this preprocessing cleaning technique, attention makes the model just too complex to be used. An argument that other attackers could invest in these more expensive and, therefore, better equipment that can run the autoencoder with attention can be nullified because we already have faster and easier techniques that need less expensive equipment (like CNN autoencoder).

The second conclusion we can draw from this chapter is that the LSTM autoencoder is, in some part, strong enough to clean noise from a trace. We have used three different countermeasures on our dataset. If we order these countermeasures in terms of how easy to remove, we would say the first one is Gaussian noise, the second one is desynchronization countermeasure, and the hardest one is random delay interrupt. We made this order based on averaging. If we averaged every dataset, the Gaussian noise countermeasure would be the easiest one to attack. If we look at Section 5.5.1, we see that for the first two of our list, the LSTM autoencoder can follow the original line quite well. We would say that for Gaussian noise, the newly generated traces' quality is around 98%, and the quality of the newly generated traces after desynchronization is around 75%. Both of these numbers are quite high and therefore satisfy an LSTM autoencoder's ability to clean traces. When the countermeasure gets harder, with random delay interrupt countermeasure, we see a 90% fit on the part where it can fit the original trace. However, after that, it is too hard and loses the trace. We would say the fit there is around 10%. The conclusion is that the LSTM autoencoder can remove noise from the traces for quite a simple countermeasure. When the countermeasure gets harder and more noise is added, the LSTM autoencoder is not a suitable option.

The last conclusion we can draw from this chapter is related to the side-channel experiments conducted in this chapter. The goal was to show that noise reduction could be accomplished and make a better side-channel attack in terms of guessing entropy. We have seen that the LSTM autoencoder can reduce the noise signature from the noisy datasets. However, if we compare the guessing entropy results on the noisy dataset and the cleaned dataset, the results are not overwhelming. The guessing

entropy for the Gaussian noise part has been increased slightly. The cleaned dataset has somewhat better results compared to the noisy dataset. Meaning that the cleaning process has worked. However, the resulting guessing entropy is not something we can work with. For the desynchronization noise, the results of the cleaned dataset are worse compared to the noisy dataset. Two possible reasons could explain this. First of all, the cleaned dataset is mostly smoothed in places where some peaks happen quite fast after each other. This smoothing process could result in less leakage present in the cleaned dataset, which results in a generalized dataset. The second reason is that desynchronization noise is only hiding and not a masking countermeasure. A hiding countermeasure is easier to break, even more, if it is added artificially afterward if we train and evaluate on the hiding dataset. The leakage is original, and the countermeasure consistent. With the third countermeasure, random delay interrupt, we combine two different countermeasures. The fact of combined countermeasures makes it harder, which can be seen from the results. The autoencoder was unable to predict clean traces, which would nicely fit on the original trace. This was also present in the guessing entropy plot, where we saw that the plot's results on clean traces were even worse than on dirty traces. We want to mention that CNN is not tweaked for the new dataset. We do see potential in cleaning by autoencoder. The guessing entropy function results can also be explained because we did not search for the proper CNN.

To conclude this chapter, we see a suitable option for LSTM autoencoders to translate and clean datasets. However, in terms of side-channel analysis, we have accurate leakage, and the dependency of this leakage could be vital for the classification of the intermediate values. We see that the noise is too much for the LSTM autoencoder. This chapter's conclusion is another example. Looking at the previous chapter, a sequential data model is not better for side-channel analysis than CNN. We have now tried different hyperparameters by our hyperparameter search in the previous chapter. This chapter looked at a more common way to use a model in many-to-many use cases. We saw somehow better results but not the one which we hoped for. Therefore we want to try one last thing before making our final conclusion. We want to look at a specific technique used in sequential data models: the embedding layer. This is something that is always used when people use a recurrent neural network. However, an embedding space does not sound like a suitable option for side-channel analysis data. Our data has less meaning than most data has that are used with sequential data models. In the next chapter, we will try to determine if this embedding layer is a suitable option for side-channel analysis and if this can be used for the whole perspective of side-channel analysis, not only when using sequential data models.

6 The Power of Embedding

In this chapter, we dive into something else. We had an in-depth look at a different natural language processing technique. We saw the need for a preprocessing technique in chapter 4; furthermore, we saw in other research that in combination with sequential data models, mostly a preprocessing technique is used. The most commonly used preprocessing technique for sequential data models is the embedding layer. Therefore, we explore the need and power of an embedding layer. Embedding is a technique commonly used in natural language processing, and there is an assumption that it could also be used in common side-channel attacks. The great benefit of embedding is that it turns data from two dimensions to three dimensions, which always happens when we use a neural network. However, with this embedding layer, the new dataset holds more information, using less memory because of the efficient way an embedding space works. So in this chapter, we took a look if it could be applicable for side-channel data. Because embedding is being used in sequential data models, we investigate how it works on the three models explored in Chapter 4, so for RNN, LSTM, and GRU. We furthermore want to investigate if this embedding layer is something new that could be used in state-of-the-art side-channel techniques. Therefore we also evaluate it on an MLP and a CNN. We want to give good advice on what should be done with this embedding layer. Therefore we use three different datasets, namely, the previously seen DPAv4 dataset, the AES dataset with random delay countermeasure, and , the ASCAD dataset.

6.1. Methodology

In this chapter, we evaluate and advise on the use of an embedding layer. We are mostly interested in side-channel analysis benefits when using an embedding layer as the first layer in a model. Therefore there is not an extensive hyperparameter search. We did run every experiment with the same values ten times and averaged it to get the model's average result. We used different setups because changing the model by adding an embedding layer could also change the setup. However, because we do not want to find the best setup but only prove the layer's effectiveness, we did not dive deeper into this. To get a more average result of the model with an embedding layer, we compare the results with state-of-the-art results. In terms of the RNN, LSTM, and GRU model, this means we compare the results with the results of Chapter 4. For the MLP and CNN, we used it as a baseline for the literature's models. Every baseline results are discussed in every section, how the embedding technique work can be found in 2.5.3. Before we start, we want to point out that using an embedding layer could have the same results as a hiding countermeasure. Therefore, using this layer could make it harder for most models; however, embedding tries to cluster the different values to groups that act similarly. We can say that it reduces the hiding countermeasure and therefore is a strength for the side-channel analysis dataset with hiding countermeasures.

6.1.1. Dataset Preparation for Usage of Embedding Layer

As input for the embedding layer, we need a dataset that is integer encoded with only positive numbers. The embedding layer makes then an encoded vector of every integer. When we create a one-hot encoded vector, the dataset's dimension is closely related to the unique values' size. If we have 5 000

different words in a dataset, which is quite fast the case in a story, our one-hot encoded vector does consist of 4999 zeros and only one 1, which is 'hiding' in this long vector length of size 5 000. Therefore we want the dataset integer encoded. For the embedding input, we need all positive integers, which is not always the case in how the datasets are formatted. Therefore we use the following techniques to deal with these problems

- Considering the ASCAD dataset, which consists of all integer values but some are also negative, we shifted the whole dataset to the number of values needed, this means when the lowest min value is -100 and the highest positive value 120. We shift the whole dataset 100 values to the right, creating a dataset that holds values from 0 to 220. This is a more desired input for the embedding layer.
- Considering the AES and DPAv4 dataset, some more preprocessing is needed because these values are already normalized and therefore are mostly float values, which are infinite precise. Therefore we choose another method. We round every value using five values behind the comma; this gives us more 'unique' values. Because now 0.0000123 and 0.00001567 are both rounded to 0.00001 and therefore become equal. Then we count the number of unique values and create a list that is holding every unique value, and therefore the list has the size of a dictionary. Then we replace every unique value in the original dataset with the index of the unique value in the created list. Because the index starts at 0, we create a new dataset, which is integer encoded and still holds the same sequence as before. What should be regarded as that the rounding of the dataset decides how accurate the new values were. Because a poor round of 2 means that all values smaller than 0.01 (e.g., 0.002) are rounded to 0.00, which reduces accuracy. However, rounding it to 20 means that every value was unique, and we do not see any recurrence of values. Therefore rounding to 5 is chosen, which is an educated guess by looking at the data. This rounded value could be optimized in further research.

Another solution to deal with the problem of the DPAv4 dataset we considered is to multiply every value by a specific factor to eliminate the decimal values. However, when we do this, we need to multiply with the smallest value, which means that, for example, when we have a value of 0.9, 0.1, and 0.00001. we need to multiply all the values with 10 000, giving us a value of 9 000,1 000, and 1. This creates a colossal dictionary size of different values, which is not optimal because it could be that many values (e.g., between 1 and 10 000) are not used. Therefore this option was not used, and we prefer to round the decimal values to a specific predefined value.

6.2. RNN, LSTM, and GRU with Embedding

For the RNN, LSTM, and GRU, we need to compete with the following baseline:

- DPAv4 dataset: Should be better than the 350 traces reached with the LSTM model, 400 with the RNN, and around 450 with the GRU model
- AES with random delay dataset: Should be better than the 300 traces reached with the LSTM. For the other two sequential data models, we do not have a baseline.
- ASCAD dataset: Here, we want to see that it can converge. If that is the case, it would be an improvement compared to previous results.

In the following section, we see that the embedding layer models do not get close to the baseline. Therefore we could argue that the model is not optimized for the different datasets that the embedding creates. Alternatively, we could play more with the dimension output of the embedding layer, which we experimented with further.

6.2.1. DPAv4 Dataset

In Figure 6.1 are the results of the experiment. Here we see the guessing entropy of the three sequential data models used in the previous chapter. The output dimension of the embedding layer was equal to 10. What can be concluded from the figure is that for GRU, the embedding does not have any effect. Even worse, it decreases performance because, without embedding, the GRU model was able to converge. The LSTM and RNN can converge to the right key; however, the results are worse than

what we have seen before. We can also see that the LSTM is doing the best job comparing the three different sequential data models. A possible explanation for worse results compared to before is because the embedding output is equal to 10. Meaning that every timestep has ten features; this was one before (which was the original feature). Changing from 1 to 10 must give some sparsity in the data. Comparing the results with the results from chapter 4, we see some consistent results. Because also here, the GRU is performing the worst of the three sequential data models.

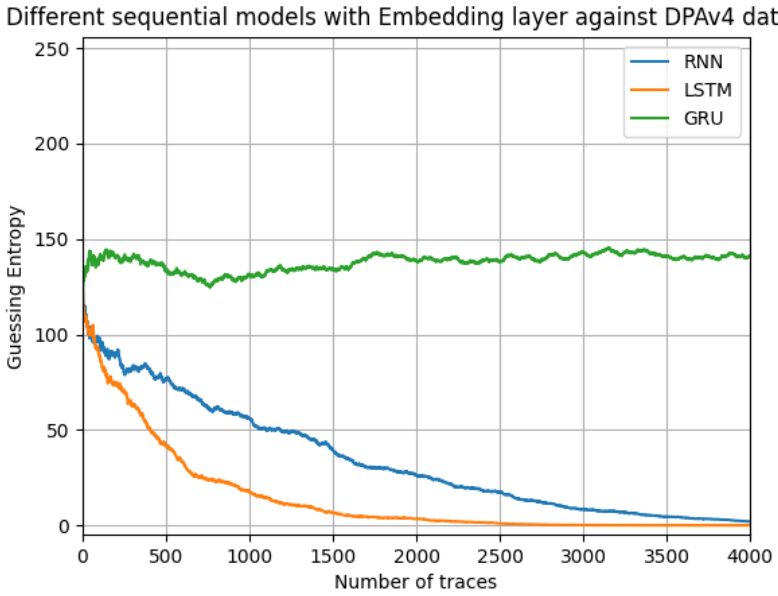


Figure 6.1: Guessing entropy of RNN, LSTM, and GRU model against the DPAv4 Dataset. At the beginning of the model is a embedding layer which creates a three dimensional data space as input with 10 units per timestep.

6.2.2. AES with Random Delay Dataset

In Figure 6.2 are the results of the experiment. We see the guessing entropy of the three sequential data models used in the previous chapter when trained and evaluated on the AES dataset with random delay countermeasures. The output dimension of the embedding layer was equal to 10. What can be concluded from the figure is that only the GRU can converge to the right key. The other models are converging but cannot reach a guessing entropy of 0 in 4 000 traces. From this, we can conclude that an embedding layer makes more sense for the hiding countermeasure than for the masking countermeasure, which makes sense when looking at the embedding layer's abilities. It does change every value and tries to optimize it. It is some hiding principle, so it should neglect a hiding countermeasure. Masking is something the embedding does not deal with from its capabilities. However, the results are the opposite than seen before, making this natural language processing method an interesting choice, which should be weighted based on the dataset's countermeasure—comparing the results with the DPAv4 dataset. We see that this is the only time the GRU model is performing better than the LSTM and the RNN. We do not see any logical reason why the GRU model performs better in this case than before.

6.2.3. ASCAD Dataset

In Figure 6.3 are the results of the experiment. Here we see the guessing entropy of the three sequential data models used in the previous chapter when trained and evaluated on the ASCAD dataset attacking the third key byte. The output dimension of the embedding layer was equal to 10. What can be concluded from the figure is that none of the models can converge to the right key. It is hard to draw any conclusions from this because the models could not break the dataset without the embedding layer, and with embedding, the layer does not change anything. We could argue that the ASCAD dataset is too hard for the models or that the models are too weak for the dataset. However, different techniques

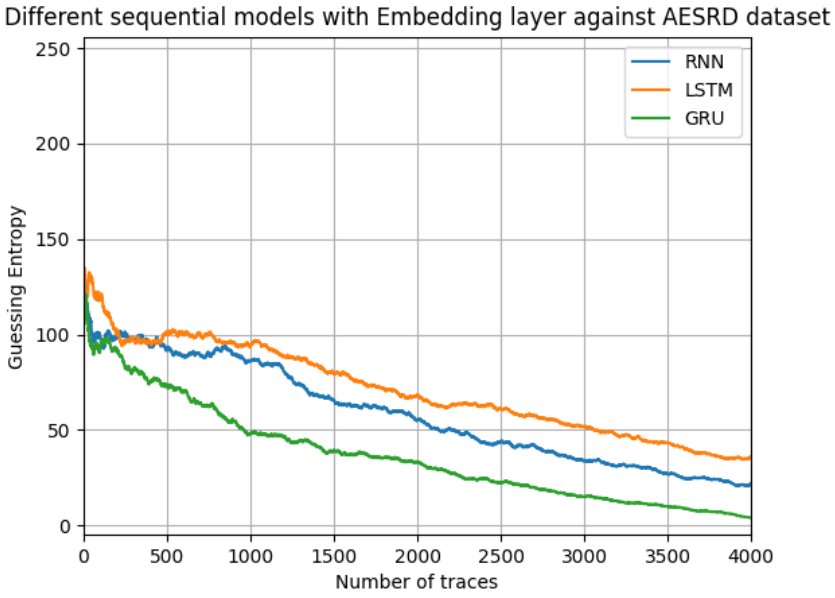


Figure 6.2: Guessing entropy of RNN, LSTM, and GRU model against the AESRD Dataset. At the beginning of the model is a embedding layer which creates a three dimensional data space as input with 10 units per timestep.

show that it does not break the encryption. From this, we can make a big conclusion about the use of sequential data models in the ASCAD dataset. The dataset is too hard for sequential data models.

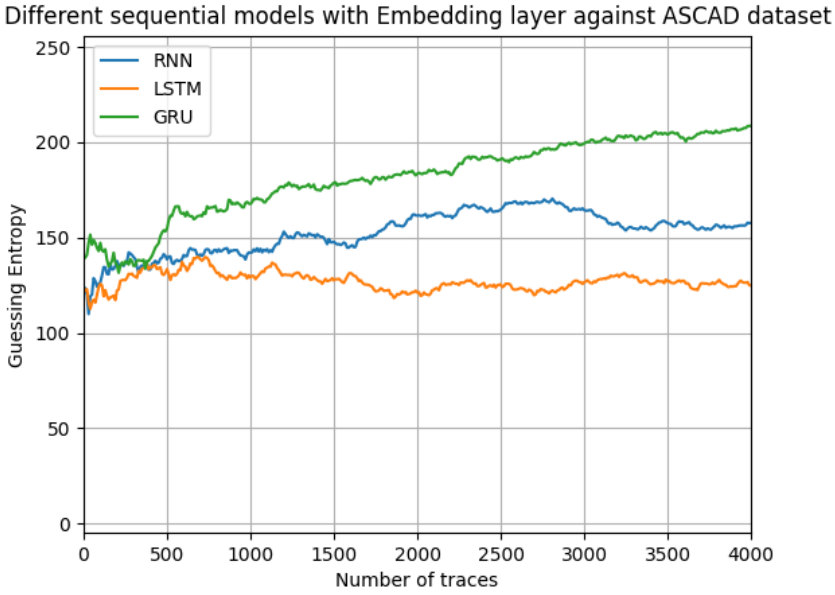


Figure 6.3: Guessing entropy of RNN, LSTM, and GRU model against the ASCAD Dataset. At the beginning of the model is a embedding layer which creates a three dimensional data space as input with 10 units per timestep.

6.3. MLP with Embedding

For MLP, we need to compete with the following baseline:

- DPAv4 dataset: According to [38], the guessing entropy reaches zero after 40 traces. The model

used has three layers, where the hidden layer has 20 neurons.

- AES with random delay dataset: According to [71], MLP can break the key when using a model with one layer, 2 000 perceptrons, and six layers with every ten perceptrons. There it can reach a guessing entropy of less than 30 within 2500 attack traces.
- ASCAD dataset: According to [71], they use the ReLU activation function, with six layers, and each layer has 200 perceptrons. The guessing entropy reaches 0 after 1 000 attack traces. The model is also used in [56].

In Figure 6.4 are the results of using three different datasets with an MLP, including embedding. What we can see is that the MLP can converge with every dataset. However, it is only able to find the correct key when considering the DPAv4 dataset. Compared with the previous section, we see that the LSTM is doing better with DPAv4 than the MLP is doing. However, both models are not optimized for the dataset they are trained on. This could explain the reason why the LSTM is stronger. Interesting is that only DPAv4 can converge. Previously we saw that that the sequential data models did better work on the AESRD. Now we see that only the DPAv4 dataset is converging. This could mean that DPAv4 is simpler to break for the MLP. Another reason could be that embedding in combination with the ASCAD and AESRD dataset is not a good combination. Because in the previous section, we saw that embedding with AESRD and ASCAD, no matter which model used, did not have any good results.

Moreover, of course, embedding is a sequential technique and should work better with sequential data models. Also, embedding is a technique to transform a two-dimensional dataset into a three-dimensional dataset. However, the input of an MLP should be two dimensional. Therefore we need a flatten layer after the embedding layer to make the input work. We could argue that this is not really convenient, and therefore does not make sense. At last, we see that the results do not come near the baseline that was set earlier.

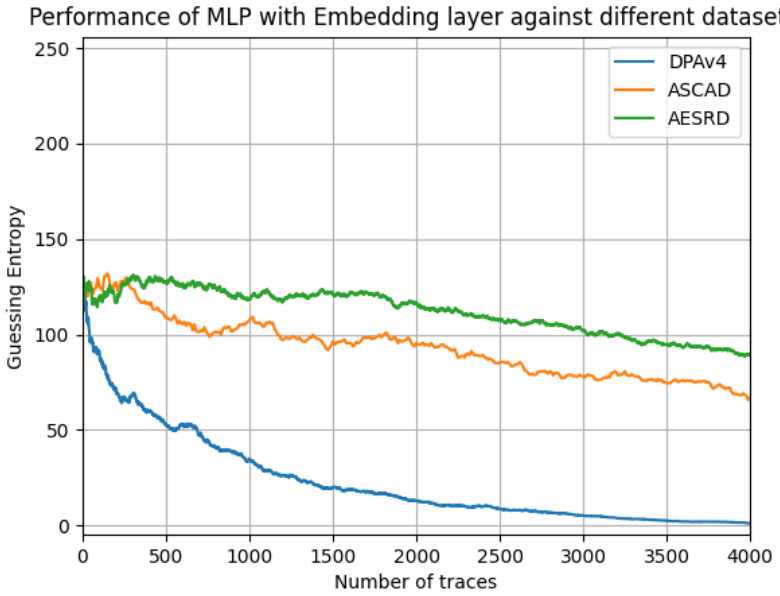


Figure 6.4: Guessing entropy of MLP model against three different datasets. At the beginning of the model is an Embedding layer, which creates a three-dimensional data space as input. With an output of 10 units per timestep.

6.4. CNN with Embedding

For CNN, we need to compete with the following baseline:

- DPAv4 dataset: according to [74], there they reach a guessing entropy of 0 within three traces. The architecture used is one convolutional layer, then a batch normalization and after that average pooling, flatten, and then two dense layers.

- AES with random delay dataset: according to [74], where he reaches a guessing entropy of 0 after four traces. The architecture used is the same as with DPAv4, but then they use four convolutional blocks instead of only one.

- ASCAD dataset: According to [56], they can find a guessing entropy of 0 after around 50 traces. They have five convolutional layers with an average pooling layer, where the convolutional layers have an increasing number of perceptrons per layer.

In Figure 6.5 are the results of each experiment. At first, we see that the CNN model is not able to converge on the ASCAD dataset. This is because CNN is more sensitive to different data input, and therefore changing the input data because of embedding makes it harder for CNN to work. Also, every timestep now is represented by ten values is an overkill of data for CNN. The other two datasets are more consistent, converging to some intermediate value but cannot reach a guessing entropy of 0 within 4 000 attack traces. It looks like CNN has problems with the huge amount of extra data that is used for training. From this aspect, we would say that an embedding layer is not a good option for CNN. Moreover, CNN has proven not to be working very well with preprocessing layers. The best we can have is to have the pure form of the data with a CNN. This embedding layer does not improve the model's performance. As of last, we would like to mention that AESRD and DPAv4 are at least converging. The ASCAD dataset is again not having any positive conclusions. This could be since ASCAD is a harder dataset and, therefore, more sensitive to an embedding layer's changes in the data.

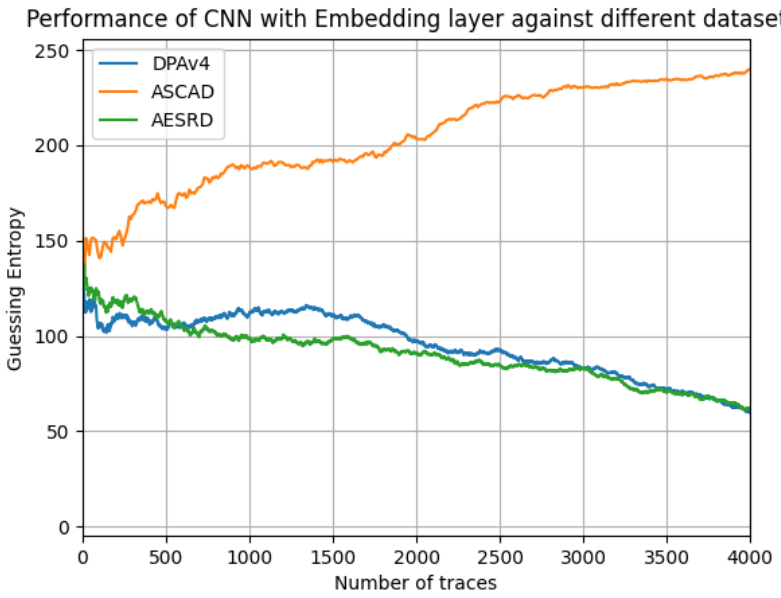


Figure 6.5: Guessing entropy of CNN model against three different datasets. At the beginning of the model is an Embedding layer, which creates a three-dimensional data space as input, with an output of 10 units per timestep.

Until now, we have seen one common recurrent issue when using this embedding layer, which we want to explore further in the next section. This problem could be due to the big dimension we used in these experiments. To see if there are any problems considering that, we experimented with different output dimensions in the next section. Using different output dimensions resulted in fewer data per time step, which could be easier for the model to learn. Therefore, we want to decrease the embedding layer's output size to 1, 2, and 5 instead of only 10, making the input data less complicated. Because of the results, we only investigated further in the DPAv4 dataset because the other datasets give terrible results, and we saw that it works better on masking than hiding countermeasures. Lastly, we only look further into the LSTM model from the sequential data models because it showed the best results.

6.5. Different Embedding Output with LSTM

Here we only experiment with the LSTM model because this sequential data model has shown to be the most potent model of the three sequential data models. We investigate different output dimensions to see how different embedding size different the function of the embedding layer and how different units per timestep influence the results.

The results of this experiment are visible in Figure 6.6. The green line results from an embedding layer, which has an output dimension equal to 5. The orange line has an embedding layer with an output dimension of 2, and the blue right uses one as the output dimension of the embedding layer. Here we see that decreasing the output dimension does decrease the results. For an output dimension of 2 and 5, we see a guessing entropy of 0 after 4 000 traces. The guessing entropy does converge for an output dimension of 1. From this experiment, we can conclude that there is for the LSTM some interest in having more features per timestep then and that it does improve the results. However, in the section 6.2.1, we also used an LSTM with DPAv4 but had an embedding output of 10. There we see better results than the results in Figure 6.6. This means that using an output dimension of 10 favors at the moment with the embedding layer when using LSTM. We can not say something about a linear better performance. This is because, with an output dimension of two and five, they are around the same. However, between one and ten, we see some linear improvement.

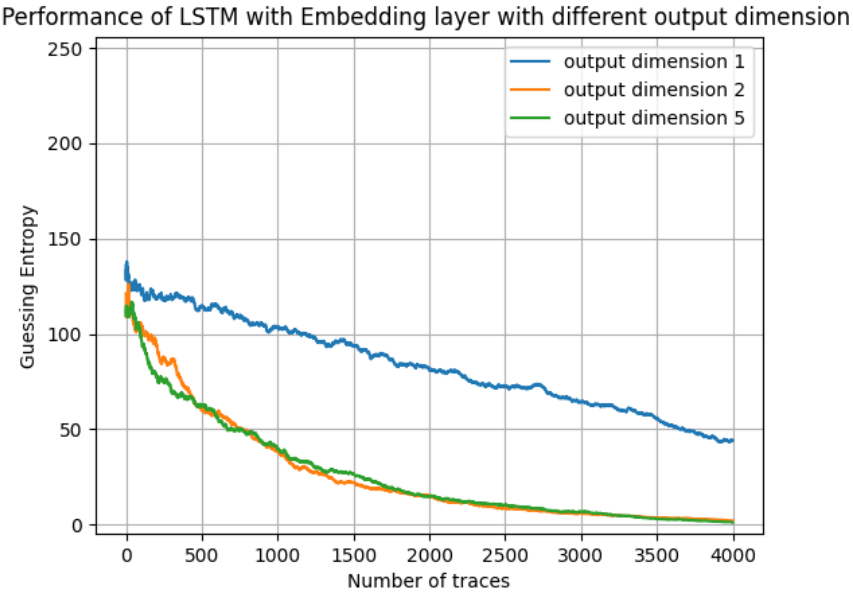


Figure 6.6: Guessing entropy of the LSTM model against DPAv4 datasets. At the beginning of the model is an Embedding layer, which creates a three-dimensional data space as input. Three different embedding outputs are shown in the figure.

6.6. Different Embedding Output with MLP

The results of this experiment are in Figure 6.7. The green line results from an embedding layer, which has an output dimension equal to five. The orange line has an embedding layer with an output dimension of two, and the blue line uses one as the output dimension of the embedding layer. Comparing the results with each other, we see something that looks like a bad run for output dimension 2. Because of that, we did a re-run, but no different results have been found. Comparing the output dimension of 1 with an output dimension of 5, we see not much difference. For the output dimension of 5, the guessing entropy reaches 0 after 3 000 traces. For the output dimension of 1, we see the guessing entropy reaching 0 after 2 000/2 500 traces. So the results are a bit better. Comparing these results with the previous section, where we had an output dimension of 10, we can conclude that the results have been improved. There we needed 4 000 traces before reaching a guessing entropy of 0. To conclude, it looks like using embedding for MLP does work; decreasing the output dimension makes the results

better than before. However, the best results are visible with an output dimension of 1. Using an output dimension of 1 does not have the power of embedding, which we expect. Using more dimensions makes the embedding special. Using only one dimension could also mean that the embedding layer maps every value to the same value.

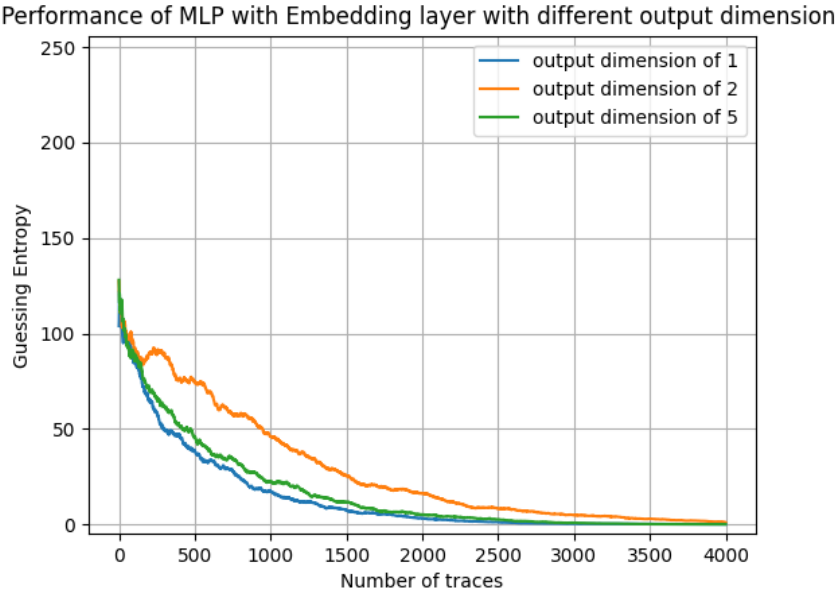


Figure 6.7: Guessing entropy of the MLP model against DPAv4 datasets. At the beginning of the model is an embedding layer, which creates a three-dimensional data space as input. Three different embedding outputs are shown in the figure.

6.7. Different Embedding Output with CNN

The results of this experiment are in Figure 6.8. The green line results from an embedding layer, which has an output dimension equal to 5. The orange line has an embedding layer with an output dimension of 2, and the blue line uses one as the output dimension of the embedding layer. Here we see how different embedding sizes are used. It appears that CNN does not work any better with different output sizes from the embedding layer per timestep. We conclude that using an embedding layer is not a good option when considering a CNN model from this experiment. This could have several reasons which are not explored further in this thesis. The most common reason could be that CNN is a shift-invariant. Meaning they can select their features and not decrease their performance when performing an attack to a hiding countermeasure. It feels like CNN is best with the most straight forward and original traces. This embedding layer does change the input a lot, and therefore the CNN does not work well with it. Another reason is that CNN is not optimized for the new data set. CNNs are known to be quite sensitive to new datasets. We did not change any hyperparameter, and the input is changed a lot. Mostly the dimension is changed, and that could also mean the architecture should be changed.

6.8. Conclusion

In this chapter, we dived into the use of an embedding layer. The reason to look into this was that having an LSTM that was not trained, only using an embedding layer, converted to the right key. This shows that the embedding layer on its own has some strength that could be used. Furthermore, we see it as a standard input layer for the natural language processing domain.

This chapter looked at using five different models, divided into three different sections—the first section where models with sequential data models; these were RNN, LSTM, and GRU. The second section had the MLP model, and the third model had the CNN model. We evaluated this embedding layer by looking at different output dimensions. This is the number of values it uses to describe the

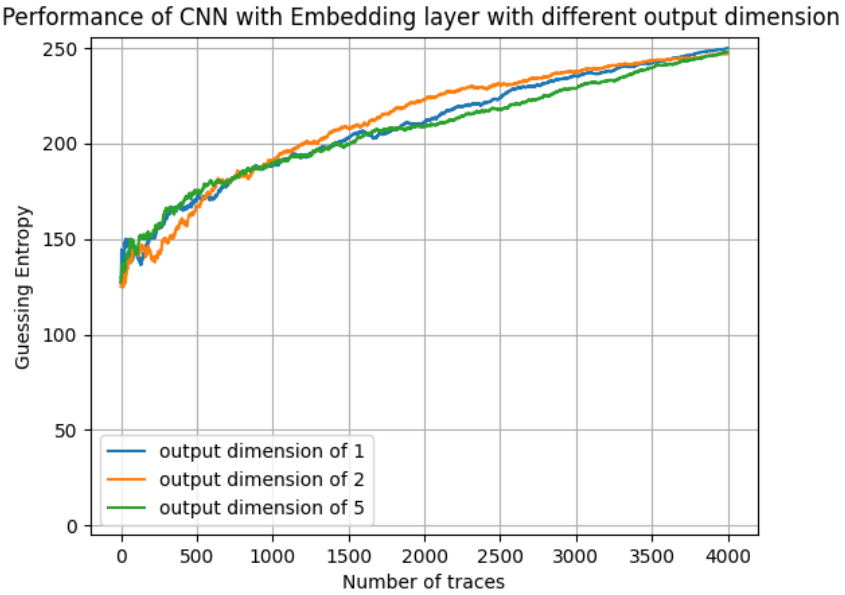


Figure 6.8: Guessing entropy of the CNN model against DPAv4 datasets. At the beginning of the model is an embedding layer, which creates a three-dimensional data space as input. Three different embedding outputs are shown in the figure.

value. In the first section, this output dimension was fixed to 10. In the section that followed the output dimension was 1, 2, and 5.

A certain timestep is represented by one value, the power at the particular time step. Embedding creates more values that describe this same value; therefore, it creates more dimensions for a particular data point. The benefit of this is that an embedding layer always creates three-dimensional data, where we usually have to change the input shape for a recurrent neural network model or a convolutional neural network. This is now being dealt with by the embedding layer. Changing the output dimension gives the user the ability to change the sparsity of the data. Because the values assigned in the embedding layer are updated continuously, we could create extra information on which values look similar when choosing two output dimensions. We can create a plot of the output of the embedding layer. There we can see which values are close to each other and which are exceptional. This could give the attacker more insight into which values are useful for the attack and which not.

Furthermore, an embedding layer could be exported and then be used for other attacks, which means that one attack can create an adequate embedding space where each integer is mapped to the right dimension and values. This attacker could do this with a big dataset, for example, the ASCAD dataset. When someone else is performing an attack with only a few traces, it could use the public to be available embedding space to map his values to the right embedding output. The embedding space is then more like a lookup table. We think this is the future for embedding layers in side-channel analysis. This lookup table for embedding space is used in natural language processing, where every language has its own embedding space. This should also be used in side-channel analysis.

In this chapter, we only looked into the use of this embedding layer for side-channel analysis. We took different baselines and experimented with the same models using an embedding layer at the beginning of the model. We can draw no positive conclusion from these experiments. This is quite logical from the fact that those results have been achieved after some years of research. We then took a more in-depth look into the use of different output sizes of the embedding layer. We saw that using an embedding layer for CNN does not improve the model at once. This could be due to the highly sensitive CNN models, so to see some results, the models should be tuned again or that a CNN does not work well with more features per timestep. For MLP and recurrent neural networks, we saw that the models were able to converge. The MLP and the sequential data model look more robust against different input. However, for the MLP, we do not see much better results. Moreover, the data has to be flattened again, making the embedding layer as the first layer less logical. The sequential data

model showed the best results with the embedding layer. After that, the MLP and then CNN are logic because the embedding layer is used more often with sequential data models. From the sequential data models, we saw the best results with the LSTM model. We also saw this in the previous chapter that the LSTM shows to be the strongest models against side-channel data. Considering different datasets, we saw that the ASCAD dataset could not get better results. It seems that this dataset is too hard for the embedding layer. Moreover, with AESRD, we saw not very exciting results. That could be explained because embedding does change the values. This means it does the same as a hiding countermeasure. Therefore we can explain why it worked better with DPAv4 because this also has a masking countermeasure, where we assume the mask is known.

We also saw better results when the output dimension decreased. Except not with the recurrent models, this could be because the output dimension indicates how many values we get to represent the original values. For the DPAv4 dataset, there were 115 000 different values in the integer encoded dataset; having only one value between 0 and 1 to represent those values in the embedding space is very hard. If we have two or three values, the options are more distinctive. That could also be why the models were behaving worse on the AESRD and ASCAD dataset when the embedding layer's output dimension was equal to 10. Because in those two datasets, the integer encoded datasets only hold 114 and 150 unique values. From this, we could say that the output dimension should be chosen carefully, taking into account what the original vector space was.

7

Conclusions and Future Work

This thesis's primary goal was to look at sequential data models in the side-channel analysis domain, which we discussed and evaluated in three different chapters. In Chapter 4, we looked into the use of sequential data models to train a neural network and perform a side-channel attack. Moreover, this chapter took a broader look into what hyperparameters should be used when using sequential data models. In Chapter 5, we looked at the autoencoder. The purpose of the chapter was to prove that an algorithm can be used to clean side-channel analysis traces, which means that a countermeasure added to original traces can be removed when a model can learn on the original and noisy traces. In Chapter 6, we looked at the embedding layer, a commonly used natural language processing technique. There we looked in what way these natural language processing techniques could be used in the side-channel analysis. We consider the data, which are the traces, as sentences with words, which is an entirely new look to side-channel analysis data. The three chapters are divided into three sections. The last section looks into this research's limitations and addresses other research topics to explore.

7.1. Evaluation of Sequential Data Models

There are many challenges when using sequential data models on side-channel analysis. Therefore, Chapter 4 investigates the use of sequential data models in side-channel analysis. This chapter's central research question is, how could RNN, LSTM, and GRU be used for the side-channel attacks? (**RQ 1.**)

To answer this question, we have experimented with the most common hyperparameters for sequential data models. We have looked into three different types of recurrent neural networks. We also experimented with different hyperparameters for every type of sequential data model. After that, we experimented with different preprocessing techniques to develop different setups of experiments for side-channel analysis and look for differences in the results. We also experimented with different countermeasures used in the traces, which we can divide into three different datasets. First, the DPAv4 where the mask was known; secondly, a dataset with AES encryption, which had a random delay countermeasure, and lastly, the ASCAD dataset. The hard part is that the trace's leakage should be sequential, which is not always the case. In the experiments, we found some promising results, but none of them better than the CNNs. The use of CNN models has been researched in the last few years and has been improved by many other researchers. Therefore, the experiments of this thesis, which performed significantly worse than CNNs, was not surprising. However, the question arises, will the results for sequential data models in side-channel analysis become better after a few more years of research? At the moment, we do not expect that to happen. Sequential data models are a suitable alternative for side-channel analysis; they can converge and find the correct key in most cases. However, when the countermeasures are getting stronger, like with the ASCAD dataset, the sequential data models cannot perform a good attack. The amount of noise present in side-channel analysis can explain this. When the dataset has a random delay countermeasure, the sequential data models are performing the best. When the dataset has a noise countermeasure (like Gaussian noise), the models perform relatively worse. This can be explained mostly due to the high noise present in the side-channel analysis traces. The performance was better when we deployed the same attack on a

different dataset without noise. The results are visible in Figure 7.1.

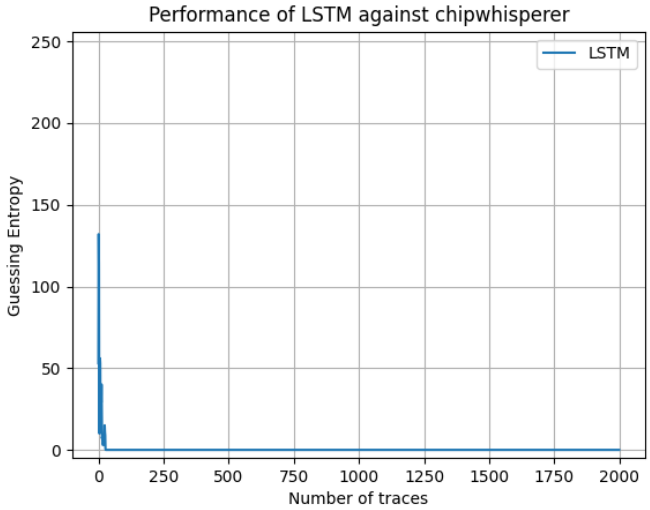


Figure 7.1: On the X-axis, the amount of traces needed to reach a GE of 0. On the Y-axis, the guessing entropy value corresponding to the amount of traces. GE reaches a value of 0 with approximately 20 traces.

If there is still some interest to keep working with these sequential data models, the use of the right hyperparameters is essential. Therefore one of the sub-questions was (**SubRQ 1.1**); what are good hyperparameters for sequential data models when dealing with side-channel analysis? In this chapter, we have taken a look at different hyperparameters. We have experimented with batch size, units, dropout, recurrent dropout, weight initializer, training size, amount of layers, and activation functions. Different values we experimented which are visible in Table 4.1. From the experiments, we conclude that it is best to use the smallest possible batch size, and the number of units should be equal to the data dimension. Dropout and recurrent dropout should be used to prevent overfitting. We used a shallow network using a small number of layers and units. For the activation function, we saw that the sigmoid function performed better than ReLU and TanH. For the weight initializer, we had the best results when using the He uniform weight initializer. The best recurrent neural network to use when performing a side-channel attack is the LSTM model. This conclusion is derived by using different hyperparameters with every model, trying every type of recurrent network on different leakage models, and looking at the attack's performance. Looking at the overall performance of every individual model, the LSTM has the best performance.

After the hyperparameter search, we started to improve the results. The main goal was to find something that would reduce the sequence length while keeping the sequential dependencies (**SubRQ 1.2**). In this thesis, we have tried multiple tricks to decrease the input sequence length. We decided to use smaller sequence lengths because the performance was the same, but the training time would decrease significantly. Therefore we tried a dataset holding only 50 features generated with Pearson correlation and a dataset holding 150 features generated with a linear regression technique. We have proven with the second technique that it is a feasible technique to use and captures most of the original traces' information. The conclusion from these experiments is that in terms of recurrent neural networks, we see that a shorter sequence length does improve the model a lot in terms of speed and that linear regression is a suitable option to do this as it still captures the relevant information. We saw that the attacks were comparable with the previous results we had in this thesis regarding the side-channel analysis. Therefore, we conclude that linear regression is a suitable option to reduce the sequence length when performing a side-channel attack. Moreover, we conclude that reducing the sequence length when using sequential data models is a must.

7.2. Denoising with Autoencoder

After the previous chapter, we concluded that sequential data models should be used for many-to-many classification. A common way to use many-to-many classification would be to translate one dataset

of traces to another dataset. This translation has been done by adding noise to the original dataset and learning an autoencoder to translate it. This setup resulted in the following main research question (**RQ 2.**): Can an LSTM autoencoder be used to denoise a trace and make a better side-channel attack in terms of the guessing entropy? The answer is yes; an LSTM autoencoder can be used to reduce noise from a trace. It can reduce noise with a hiding characteristic; then, the side-channel attack will be better. The first sub research question (**SubRQ 2.1**) is how well do the denoised traces fit the original traces? This fitting depends on which type of noise is used. In this thesis, we have experimented with two different types of noise. We saw that a hiding countermeasure was easily removed when the noise consists of Gaussian noise. The random delay was more challenging for the autoencoder to remove from the traces. We concluded that an autoencoder is better at removing noise when it is not shifted. When a leakage value is shifted, the difference between the original value and the shifted value differs at every timestep. Because of the trace's sequential shape (sinus shape), they even cross each other, and then the difference gets negative. Considering Gaussian noise, the difference is always between the same limit (min and max value of Gaussian distribution) and not time-dependent. The third dataset we used had both random delay and Gaussian noise. There we saw quite clearly that it was quite hard for the LSTM autoencoder to denoise the dataset. Acquiring denoising results, we wonder in what terms the significant leakage of side-channel analysis trace has been brought back in the clean trace or has been removed from the trace. Therefore we want to evaluate the traces, which resulted in the second sub research question (**SubRQ 2.2**). How do denoised traces perform compared to original traces when performing a side-channel attack with CNN and evaluate using guessing entropy function? We have performed an attack with the noisy (artificially added noise) dataset and the cleaned (denoised) dataset to answer this question. With the first attack, we retrieved a baseline about what would happen after the noise is added. The second attack is then compared with the baseline. Here we can draw the same conclusions that we already made when we only looked at the traces. The dataset with Gaussian noise is performing slightly better than the baseline model. So here we see a performance increase compared to the baseline. With random delay countermeasures and combined with Gaussian noise, the other two models perform far worse than the baseline. This means the autoencoder could reproduce most of the trace but could not precisely reproduce the most critical leakage. We prefer to have random delay countermeasures instead of Gaussian noise when performing an attack, but we prefer to have Gaussian noise when removing noise.

7.3. The Power of Embedding

To conclude sequential data models, we want to try a specific layer commonly used in the natural language processing domain. Therefore we want to explore the embedding layer. Therefore the question (**RQ 3**) arises: Should embedding be used as a preprocessing method in the side-channel analysis? Embedding could be used as a preprocessing method. We saw that embedding for CNN does not work well. This can be explained because CNN filters its values and has proven that it does not work well with data preprocessing. However, with a multilayer perceptron and a sequential data model, we saw better results. Unfortunately, it does not perform better than the state-of-the-art results, but we also consider that the model's hyperparameters have not been adjusted to the embedding layer. Furthermore, we want to address that when using embedding, the model tries to create a vector space by combining features. Because of the noise in traces, it is harder to divide the traces into clear separate groups, which happens when using words. This could explain why it was performing worse, and we expect that it would do even better when we have a good feature selection. Therefore we conclude that embedding is a good alternative as the first layer when training a sequential data model for side-channel analysis.

However, the output dimension of the embedding layer should be chosen carefully. Therefore the following subquestion was part of this chapter (**SubRQ 3.1**); what output dimension should be used when using an embedding layer? The output dimension decides the sparsity of the data. Having more features per timestep does influence the ideal output dimension. In our case, we only have one feature per timestep. Therefore we should keep the output dimension close to the number of features. In the experiments, we saw a linear increase between using the dimension output of one and ten, but a decrease using 2 and 5. Using one as an output dimension does not do much with the embedding space; therefore, we prefer to use ten as an output dimension.

7.4. Future Work

This work does an in-depth study of the use of sequential data models. There are always possible future works to consider. In the section, we explain the limitations and the future work for the topics covered in this thesis.

The most significant limitation of this research is that we have worked with networks from the deep learning domain. Deep learning is considered a black-box algorithm, and therefore there is no real understanding of how the network is learning. Since we used the LSTM, RNN, and GRU models in this work, which are part of the deep learning domain, we have no guarantee that the results are reproducible. We have tried different hyperparameters, different models, and different datasets with different characteristics to the best of our knowledge. We are making this research the first guide into a good setup for side-channel analysis with sequential data models and, therefore, a good starting point. However, some minor tweaks or different hyperparameters could mean completely different results. Furthermore, we limited running time for a single model to a maximum of 48 hours. This could have been more, but we take more than mentioned as not feasible from the aspect of a side-channel attack since there are already better models that perform an attack in less time. The last limitation comes with the embedding layer. In the best possible option, we have a trained embedding space from the complete set of data. The data we use for the model can be a subset of this data used to create the embedding space. In our case, where the embedding space was a completely new suggestion for side-channel analysis, we had to train our embedding space with our data. This means our dataset was equal to the size of the embedding space. Meaning we could not use the effect of the embedding layer at a full capacity.

In this thesis, we have addressed three research questions and answered them in different chapters. However, because of the limitations and the scope of the project, not everything has been researched. In the following paragraph, we address future work that could be done. We start by suggesting that we should not dive any deeper into looking for the right hyperparameters, considering the characteristics used with datasets in this thesis, which means that we have tried almost every possible setup, with a wide variety of possible values. If there is another dataset with new characteristics for side-channel analysis, we should look into sequential data models. However, we think that preprocessing and feature selection is the key to find a better performing model in terms of side-channel attacks with sequential data models. Therefore we advise taking a more in-depth look into good performing preprocessing techniques. This is not commonly done with side-channel analysis; based on the results, we think it is a relevant topic to explore. More importantly, we would address that a good feature selection should be used.

Furthermore, we see three additional topics that can be explored considering the sequential data models. The first thing is to combine them with CNN and then use it for side-channel analysis. Secondly, we can have more features per timestep. Sequential data models are the ideal networks to use with more units, and therefore we need more features per timestep. We now use the chip's leakage, but maybe it works even better if we also use leakage of other chips or other usages inside the hardware under attack. By doing this, we have more data that can be used for the sequential data model. A third suitable option for sequential data models is to use the attention technique. This new research direction could be used with side-channel analysis, especially with sequential data models.

After the research of Chapter 5, we would like to propose an algorithm to clean traces. Think of a cloud-based solution where we can send our noisy traces with a hiding countermeasure, and there is a cleaning algorithm that gives us cleaned traces back on which we can make a faster and better attack than before. We want an algorithm that can remove countermeasures and has as input to what extend the countermeasure is used; for example, when removing random delay countermeasure, say what we expect to be the maximum delay. With this, we get a generic and powerful cleaning algorithm accessible to everyone. We think this is possible and therefore encourages us to do more research on the possibilities. On the other hand, we can also add different or more complex countermeasures to the traces that make this cloud-based cleaning algorithm useless. After the experiments in Chapter 6, we come with the following future works that are possible to investigate further. The first is that in this chapter, we have taken a look at the use of an embedding layer. However, the models have not been optimized for the different outputs from the embedding layer; this could be done in further work. Also, playing more with the embedding layer, changing the output dimension's size could influence the research result. Lastly, which has also been addressed in the limitations of this research, is that a universal embedding space is built with datasets using the same encryption and leakage. This is

something that, after a good build, can be used all around the world. In the same way, there is an embedding space for every language in the world. Creating this for side-channel analysis data would considerably improve the embedding technique and its strength. If this embedding space is well trained, we could even add extra data points and see which side of the space this data point belongs to. This could give the attacker information in what way this point is holding leakage information.

Bibliography

[1] Télécom ParisTech DPAv4 - Download. URL http://www.dpacontest.org/v4/.

[2] HPC Cluster. URL https://login.hpc.tudelft.nl/.

[3] 7 Types of Activation Functions in Neural Networks: How to Choose? URL https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/.

[4] ANSSI-FR/ASCAD: Side Channels Analysis and Deep Learning. URL https://github.com/ANSSI-FR/ASCAD.

[5] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, sep 2015.

[6] Timo Bartkewitz and Kerstin Lemke-Rust. Efficient Template Attacks Based on Probabilistic Multi-class Support Vector Machines. pages 263–276. Springer, Berlin, Heidelberg, nov 2013. doi: 10.1007/978-3-642-37288-9_18. URL http://link.springer.com/10.1007/978-3-642-37288-9{_}18.

[7] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. 2011.

[8] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3156, pages 16–29. Springer, Berlin, Heidelberg, 2004. doi: 10.1007/978-3-540-28632-5_2. URL http://link.springer.com/10.1007/978-3-540-28632-5{_}2.

[9] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. pages 45–68. Springer, Cham, 2017. doi: 10.1007/978-3-319-66787-4_3. URL http://link.springer.com/10.1007/978-3-319-66787-4{_}3.

[10] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. pages 13–28. Springer, Berlin, Heidelberg, 2003. doi: 10.1007/3-540-36400-5_3. URL http://link.springer.com/10.1007/3-540-36400-5{_}3.

[11] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long Short-Term Memory-Networks for Machine Reading. *Proceedings of the 30th Annual Conference of the Japanese Society for Artificial Intelligence*, 2(3):2–4, jan 2016. URL http://arxiv.org/abs/1601.06733.

[12] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. pages 103–111, sep 2014. URL http://arxiv.org/abs/1409.1259.

[13] Jean Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5747 LNCS, pages 156–170. Springer, Berlin, Heidelberg, 2009. ISBN 364204137X. doi: 10.1007/978-3-642-04138-9_12.

[14] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. pages 277–284. Springer, Berlin, Heidelberg, 2000. doi: 10.1007/10721064_26. URL http://link.springer.com/10.1007/10721064{_}26.

[15] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis: A generic side-channel distinguisher. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5154 LNCS, pages 426–442. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 354085052X. doi: 10.1007/978-3-540-85053-3_27. URL http://link.springer.com/10.1007/978-3-540-85053-3{_}27.

[16] Nihal Fatma Güler, Elif Derya Übeyli, and İnan Güler. Recurrent neural networks employing Lyapunov exponents for EEG signals classification. *Expert Systems with Applications*, 29(3): 506–514, oct 2005. ISSN 0957-4174. doi: 10.1016/J.ESWA.2005.04.011. URL https://www.sciencedirect.com/science/article/pii/S0957417405000679.

[17] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29 (2):147–160, 1950. ISSN 15387305. doi: 10.1002/j.1538-7305.1950.tb00463.x.

[18] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Profiled Power Analysis Attacks Using Convolutional Neural Networks with Domain Knowledge. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11349 LNCS, pages 479–498. Springer, Cham, aug 2019. ISBN 9783030109691. doi: 10.1007/978-3-030-10970-7_22. URL http://link.springer.com/10.1007/978-3-030-10970-7{_}22.

[19] Annelie Heuser and Michael Zohner. Intelligent machine homicide: Breaking cryptographic devices using support vector machines. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7275 LNCS, pages 249–264. Springer, Berlin, Heidelberg, 2012. ISBN 9783642299117. doi: 10.1007/978-3-642-29912-4_18. URL https://link.springer.com/chapter/10.1007/978-3-642-29912-4{_}18.

[20] Annelie Heuser, Stjepan Picek, Sylvain Guilley, and Nele Mentens. Side-Channel Analysis of Lightweight Ciphers: Does Lightweight Equal Easy? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10155 LNCS, pages 91–104. Springer Verlag, 2017. doi: 10.1007/978-3-319-62024-4_7. URL http://link.springer.com/10.1007/978-3-319-62024-4{_}7.

[21] Annelie Heuser, Stjepan Picek, Sylvain Guilley, and Nele Mentens. Lightweight Ciphers and their Side-channel Resilience. *IEEE Transactions on Computers*, sep 2017. ISSN 15579956. doi: 10.1109/TC.2017.2757921.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9 (8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735.

[23] 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):293–302, dec 2011. ISSN 21908508. doi: 10.1007/s13389-011-0023-x. URL http://link.springer.com/10.1007/s13389-011-0023-x.

[24] 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):293–302, dec 2011. ISSN 21908508. doi: 10.1007/s13389-011-0023-x. URL http://link.springer.com/10.1007/s13389-011-0023-x.

[25] 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):293–302, dec 2011. ISSN 2190-8508. doi: 10.1007/s13389-011-0023-x. URL http://link.springer.com/10.1007/s13389-011-0023-x.

[26] Po-Yao Huang, Frederick Liu, Sz-Rung Shiang, Jean Oh, and Chris Dyer. Attention-based Multimodal Neural Machine Translation. Technical report.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning, ICML 2015*, volume 1, pages 448–456. International Machine Learning Society (IMLS), feb 2015. ISBN 9781510810587.

[28] Shigeki Karita, Nanxin Chen, Tomoki Hayashi, Takaaki Hori, Hirofumi Inaguma, Ziyan Jiang, Masao Someki, Nelson Enrique Yalta Soplin, Ryuichi Yamamoto, Xiaofei Wang, Shinji Watanabe, Takenori Yoshimura, and Wangyou Zhang. A Comparative Study on Transformer vs RNN in Speech Applications. 9(4), 2019. URL http://arxiv.org/abs/1909.06317.

[29] 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. *tCHES 2019*, 2019(3):148–179, may 2019. doi: 10.13154/tches.v2019.i3.148-179.

[30] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. pages 388–397. Springer, Berlin, Heidelberg, 1999. doi: 10.1007/3-540-48405-1_25. URL http://link.springer.com/10.1007/3-540-48405-1{_}25.

[31] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, apr 2011. ISSN 21908508. doi: 10.1007/s13389-011-0006-y. URL http://link.springer.com/10.1007/s13389-011-0006-y.

[32] François Koeune and François-Xavier Standaert. A Tutorial on Physical Security and Side-Channel Attacks. pages 78–108. Springer, Berlin, Heidelberg, 2005. doi: 10.1007/11554578_3. URL https://link.springer.com/chapter/10.1007/11554578{_}3.

[33] Shiu Kumar, Alok Sharma, and Tatsuhiko Tsunoda. Brain wave classification using long short-term memory network based OPTICAL predictor. *Scientific Reports*, 9(1):1–13, dec 2019. ISSN 20452322. doi: 10.1038/s41598-019-45605-1.

[34] Shiu Kumar, Alok Sharma, and Tatsuhiko Tsunoda. Brain wave classification using long short-term memory network based OPTICAL predictor. *Scientific Reports*, 9(1):1–13, dec 2019. ISSN 20452322. doi: 10.1038/s41598-019-45605-1. URL https://www.nature.com/articles/s41598-019-45605-1.

[35] César Laurent, Gabriel Pereyra, Philémon Brakel, Ying Zhang, and Yoshua Bengio. Batch Normalized Recurrent Neural Networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2016-May:2657–2661, oct 2015. URL http://arxiv.org/abs/1510.01378.

[36] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. apr 2015. URL http://arxiv.org/abs/1504.00941.

[37] 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 *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9064, pages 20–33. Springer Verlag, 2015. doi: 10.1007/978-3-319-21476-4_2. URL http://link.springer.com/10.1007/978-3-319-21476-4{_}2.

[38] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10076 LNCS, pages 3–26. Springer, Cham, dec 2016. ISBN 9783319494449. doi: 10.1007/978-3-319-49445-6_1. URL http://link.springer.com/10.1007/978-3-319-49445-6{_}1.

[39] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking Cryptographic Implementations Using Deep Learning Techniques. pages 3–26. Springer, Cham, dec 2016. doi: 10.1007/978-3-319-49445-6_1. URL http://link.springer.com/10.1007/978-3-319-49445-6{_}1.

[40] Tal G Malkin and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. pages 1–32, 2009.

[41] Thomas Mangard, Stefan, Oswald, Elisabeth, Popp. Hiding. In *Power Analysis Attacks*, pages 167–199. Springer US, Boston, MA, 2007. doi: 10.1007/978-0-387-38162-6_7. URL http://link.springer.com/10.1007/978-0-387-38162-6{_}7.

[42] Thomas Mangard, Stefan, Oswald, Elisabeth, Popp. Masking. In *Power Analysis Attacks*, pages 223–244. Springer US, Boston, MA, 2007. doi: 10.1007/978-0-387-38162-6_9. URL http://link.springer.com/10.1007/978-0-387-38162-6{_}9.

[43] Marcel_medwed. countermeasures, 2013. URL https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/marcel{_}medwed{_}countermeasures.pdf.

[44] Massoud Masoumi, Pouya Habibi, and Mohammad Jadidi. Efficient implementation of masked AES on Side-Channel Attack Standard Evaluation Board. In *International Conference on Information Society, i-Society 2015*, pages 151–156. Institute of Electrical and Electronics Engineers Inc., dec 2015. ISBN 9781908320483. doi: 10.1109/i-Society.2015.7366878.

[45] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, aug 2018.

[46] Amir Moradi, Sylvain Guilley, and Annelie Heuser. Detecting hidden leakages. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8479 LNCS, pages 324–342. Springer Verlag, 2014. ISBN 9783319075358. doi: 10.1007/978-3-319-07536-5_20.

[47] Mohammad Ali Naderi and Homayoun Mahdavi-Nasab. Analysis and classification of EEG signals using spectral analysis and recurrent neural networks. In *2010 17th Iranian Conference of Biomedical Engineering (ICBME)*, pages 1–4. IEEE, nov 2010. ISBN 978-1-4244-7483-7. doi: 10.1109/ICBME.2010.5704931. URL http://ieeexplore.ieee.org/document/5704931/.

[48] P. Nagabushanam, S. Thomas George, and S. Radha. EEG signal classification using LSTM and improved neural network algorithms. *Soft Computing*, pages 1–23, nov 2019. ISSN 14337479. doi: 10.1007/s00500-019-04515-0.

[49] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. nov 2018. URL http://arxiv.org/abs/1811.03378.

[50] A.A. Petrosian, D.V. Prokhorov, W. Lajara-Nanson, and R.B. Schiffer. Recurrent neural network-based approach for early recognition of Alzheimer's disease in EEG. *Clinical Neurophysiology*, 112(8):1378–1387, aug 2001. ISSN 1388-2457. doi: 10.1016/S1388-2457(01)00579-X. URL https://www.sciencedirect.com/science/article/pii/S138824570100579X.

[51] Arthur Petrosian, Danil Prokhorov, Richard Homan, Richard Dasheiff, and Donald Wunsch. Recurrent neural network based prediction of epileptic seizures in intra- and extracranial EEG. *Neurocomputing*, 30(1-4):201–218, jan 2000. ISSN 09252312. doi: 10.1016/S0925-2312(99)00126-5. URL https://www.sciencedirect.com/science/article/pii/S0925231299001265.

[52] S ; Picek, A ; Heuser, A ; Jovic, S ; Bhasin, and F Regazzoni. The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. 2019. doi: 10.13154/tches.v2019.i1.209-237. URL https://doi.org/10.13154/tches.v2019.i1.209-237.

[53] Stjepan Picek, Annelie Heuser, Alan Jovic, Simone A. Ludwig, Sylvain Guilley, Domagoj Jakobovic, and Nele Mentens. Side-channel analysis and machine learning: A practical perspective. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2017-May, pages 4095–4102. Institute of Electrical and Electronics Engineers Inc., jun 2017. ISBN 9781509061815. doi: 10.1109/IJCNN.2017.7966373.

[54] Stjepan Picek, Annelie Heuser, Alan Jovic, and Lejla Batina. A Systematic Evaluation of Profiling through Focused Feature Selection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2802–2815, dec 2019. ISSN 15579999. doi: 10.1109/TVLSI.2019.2937365.

[55] Emmanuel Prouff and Robert Mcevoy. First-Order Side-Channel Attacks on the Permutation Tables Countermeasure-Extended Version. Technical report, 2010.

[56] 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. *CoRR*, pages 1–45, 2018.

[57] 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. *CoRR*, pages 1–45, 2018.

[58] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. feb 2014. URL http://arxiv.org/abs/1402.1128.

[59] Werner Schindler, Kerstin Lemke, and Christof Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In *Lecture Notes in Computer Science*, volume 3659, pages 30–46. Springer Verlag, 2005. doi: 10.1007/11545262_3. URL http://link.springer.com/10.1007/11545262{_}3.

[60] Shijie Song, Kaiyan Chen, and Yang Zhang. Overview of Side Channel Cipher Analysis Based on Deep Learning. *Journal of Physics: Conference Series*, 1213:022013, 2019. ISSN 1742-6588. doi: 10.1088/1742-6596/1213/2/022013.

[61] François Xavier Standaert, Eric Peeters, and Jean Jacques Quisquater. On the masking countermeasure and higher-order power analysis attacks. In *International Conference on Information Technology: Coding and Computing, ITCC*, volume 1, pages 562–567, 2005. ISBN 0769523153. doi: 10.1109/itcc.2005.213.

[62] François Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5479 LNCS, pages 443–461. Springer, Berlin, Heidelberg, 2009. ISBN 3642010008. doi: 10.1007/978-3-642-01001-9_26.

[63] A. M. Turing. COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, oct 1950. doi: 10.1093/MIND. URL https://academic.oup.com/mind/article/LIX/236/433/986238.

[64] Elif Derya Übeyli. Analysis of EEG signals by implementing eigenvector methods/recurrent neural networks. *Digital Signal Processing: A Review Journal*, 19(1):134–143, jan 2009. ISSN 10512004. doi: 10.1016/j.dsp.2008.07.007. URL https://www.sciencedirect.com/science/article/pii/S1051200408001243.

[65] D van der Valk, S Picek, and S Bhasin. Kilroy was here: The First Step Towards Explainability of Neural Networks in Profiled Side-channel Analysis. *Pdfs.Semanticscholar.Org*, 2020. URL https://pdfs.semanticscholar.org/9eef/0f059d9c28c3c376ff618f4af925d7b5679c.pdf.

[66] Ashish Vaswani, Google Brain, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. Technical report, 2017.

[67] Jiang Wang, Yi Yang, Junhua Mao, Zhiheng Huang, Chang Huang, and Wei Xu. CNN-RNN: A Unified Framework for Multi-Label Image Classification, 2016. URL https://www.cv-foundation.org/openaccess/content{_}cvpr{_}2016/html/Wang{_}CNN-RNN{_}A{_}Unified{_}CVPR{_}2016{_}paper.html.

[68] Ping Wang, Aimin Jiang, Xiaofeng Liu, Jing Shang, and Li Zhang. LSTM-based EEG classification in motor imagery tasks. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 26(11):2086–2095, 2018. ISSN 15344320. doi: 10.1109/TNSRE.2018.2876129.

[69] Ping Wang, Aimin Jiang, Xiaofeng Liu, Jing Shang, and Li Zhang. LSTM-based EEG classification in motor imagery tasks. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 26(11):2086–2095, 2018. ISSN 15344320. doi: 10.1109/TNSRE.2018.2876129.

[70] Man Wei, Danping Shi, Siwei Sun, Peng Wang, and Lei Hu. Convolutional Neural Network Based Side-Channel Attacks with Customized Filters. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11999 LNCS, pages 799–813. Springer, dec 2020. ISBN 9783030415785. doi: 10.1007/978-3-030-41579-2_46.

[71] Léo Weissbart, Stjepan Picek, and Lejla Batina. On the Performance of Multilayer Perceptron in Profiling Side-channel Analysis. Technical report.

[72] Lichao Wu and Stjepan Picek. Remove Some Noise : On Pre-processing of Side-channel Measurements with Autoencoders. *Cryptology ePrint Archive*, (Report 2019/1474), 2019. URL https://eprint.iacr.org/2019/1474.

[73] Zhengzheng Xing, Jian Pei, and Eamonn Keogh. A brief survey on sequence classification. *ACM SIGKDD Explorations Newsletter*, 12(1):40–48, nov 2010. ISSN 1931-0145. doi: 10.1145/1882471.1882478. URL https://dl.acm.org/doi/10.1145/1882471.1882478.

[74] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for Efficient CNN Architectures in Profiling Attacks. *tCHES 2020*, 2020(1):1–36, 2020. doi: 10.13154/tches.v2020.i1.1-36.

[75] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. nov 2016. URL http://arxiv.org/abs/1611.01578.

A

Implementation Details

All the experiments conducted in this thesis are run on the High-Performance Cluster(HPC) [2] of the Delft University of Technology. All the networks have been trained and evaluated on this HPC. The HPC is running Linux CentOS 7. The Nodes on the cluster are equipped with GTX 1080 Ti graphic cards with 11 GB of memory and 3584 processing cores. All the experiments' code is implemented in Python 3.6.8 using TensorFlow (GPU compatible) version 2.1.0. The TensorFlow version is built against CUDA 10.1, which uses CUDA Deep neural Network library (cuDNN) version 10.1-7.6.0.64.

A.1. Reproducibility

For the experiments' reproducibility, everything is explained in detail in the representative chapter's methodology section. In this section is the pseudocode of some of these experiments.
In listing A.1 the source code snippets for the experiments for Chapter 4. In listing A.2 the source code snippets for the experiments for Chapter 5. In listing A.3 the source code snippets for the experiments for Chapter 6.

Listing A.1: Python code chapter 4. The X in the code is filled in differently per experiment.

```
size_dataset = 8\,000
dataset = np.load(input_data)
key_data = np.load(key_data)
x_train = dataset[:size_dataset]
x_train = preprocessing.normalize(x_train)
y_train = key_data[:size_dataset]

x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
y_train = np.reshape(y_trainlid, ( y_train.shape[0], y_train.shape[1], 1))

batch_size = X
Epochs = X
Dropout = X
activation = "softmax"
optimizer = Adam(lr=0.001)
recurrent_dropout = X
units = X

x_t, x_valid, y_train, y_valid = train_test_split(x_train, y_train,test_size=0.2)

y_train = to_categorical(y_train)
y_valid = to_categorical(y_valid)
```

```
model = Sequential()
model.add(LSTM(units=units, dropout=dropout,
 recurrent_dropout=recurrent_dropout, return_sequences=True))
model.add(LSTM(units=units, dropout=dropout,
 recurrent_dropout=recurrent_dropout, return_sequences=True))
model.add(LSTM(units=units, dropout=dropout,
 recurrent_dropout=recurrent_dropout, return_sequences=True))
model.add(LSTM(units=units, dropout=dropout,
recurrent_dropout=recurrent_dropout))

model.add(Dense(256, activation=activation))
model.summary()
model.compile(loss='categorical_crossentropy',
optimizer=optimizer,
metrics=['accuracy'])

history = model.fit(x_t, y_train,validation_data=(x_valid, y_valid),
epochs=epochs, batch_size=batch_size, verbose=2)
score = model.evaluate(x_valid, y_valid, verbose=2)
predictions = model.predict(x_valid)

result = calc_ge(predictions)
```

Listing A.2: Python code chapter 5. The X in the code is filled in differently per experiment.

```
X = np.load(one_hot_encode_noisy)
Y = np.load(one_hot_encode_original)

batch_size = 100
X, x_valid, Y, y_valid = train_test_split(X,Y,test_size=0.5)

X = np.reshape(X, (X.shape[0], X.shape[1], 1))
x_valid = np.reshape(x_valid, ( x_valid.shape[0], x_valid.shape[1], 1))
Y = np.reshape(Y, (Y.shape[0], Y.shape[1], 1))
y_valid = np.reshape(y_valid, ( y_valid.shape[0], y_valid.shape[1], 1))

# configure problem
n_features = 112
n_timesteps_in = 700
n_timesteps_out = 700

model = Sequential()
model.add(LSTM(units=256,dropout=0.2, input_shape=(X.shape[1],1)
,return_sequences=True ))
model.add(LSTM(units=128,dropout=0.2, input_shape=(X.shape[1], 1),
return_sequences=True ))
model.add(LSTM(units=64,dropout=0.2, input_shape=(X.shape[1],  1),
return_sequences=True ))
model.add(LSTM(units=32,dropout=0.2, input_shape=(X.shape[1],  1),
return_sequences=False ))
model.add(RepeatVector(700))
model.add(LSTM(units=32, return_sequences=True))
model.add(LSTM(units=64, return_sequences=True))
model.add(LSTM(units=128, return_sequences=True))
```

```
model.add(LSTM(units=256, return_sequences=True))
model.compile(optimizer='adam', loss='mse')
model.fit(X, Y,  batch_size=batch_size, epochs=200, verbose=2)


d3 = model.predict(x_valid, verbose=2)


new_array = np.zeros((d3.shape[0],d3.shape[1]))
for x in range(0,d3.shape[0]):
        for y in range(0,d3[x].shape[0]):
                temp = np.argmax(d3[x,y])
                new_array[x,y]=temp

new_array is used to train CNN model and evaluated
like the code used in ASCAD paper from prouff.
```

Listing A.3: Python code chapter 6. The X in the code is filled in differently per experiment.

```
size_dataset = 8\,000
dataset = np.load(input_data)
key_data = np.load(key_data)
x_train = dataset[:size_dataset]
x_train = preprocessing.normalize(x_train)
y_train = key_data[:size_dataset]

batch_size = X
Epochs = X
Dropout = X
activation = "softmax"
optimizer = Adam(lr=0.001)
recurrent_dropout = X
units = X
output_dimension  = X

x_t, x_valid, y_train, y_valid = train_test_split(x_train, y_train,test_size=0.2)

y_train = to_categorical(y_train)
y_valid = to_categorical(y_valid)

model = Sequential()
model.add(Embedding(input=unique.size,output=output_dimension))
model.add(LSTM(units=units, dropout=dropout,
recurrent_dropout=recurrent_dropout, return_sequences=True))
model.add(LSTM(units=units, dropout=dropout,
recurrent_dropout=recurrent_dropout, return_sequences=True))
model.add(LSTM(units=units, dropout=dropout,
recurrent_dropout=recurrent_dropout, return_sequences=True))
model.add(LSTM(units=units, dropout=dropout, recurrent_dropout=recurrent_dropout))

model.add(Dense(256, activation=activation))
model.summary()
model.compile(loss='categorical_crossentropy',
optimizer=optimizer,
metrics=['accuracy'])
```

```
history = model.fit(x_t, y_train,validation_data=(x_valid, y_valid),
epochs=epochs, batch_size=batch_size, verbose=2)
score = model.evaluate(x_valid, y_valid, verbose=2)
predictions = model.predict(x_valid)

result = calc_ge(predictions)
```

B

List of Abbreviations

Abbreviations	Full Term	Found in
DES	Data Encryption Standard	2.1
AES	Advanced Encryption Standard	2.1
SCA	Side-Channel Attack	2.2
RNN	Recurrent Neural Networks	2.4.1
ML	Machine Learning	2.3
LSTM	Long Short-Term Memory	2.4.2
GRU	Gated Recurrent Unit	2.4.3
GE	Guessing Entropy	2.2.4
ASCAD	ANSSI SCA Database	2.6.3
HW	Hamming Weight	2.2.5
ID	Intermediate value	2.2.5
AE	Autoencoder	2.4.5
CNN	Convolutional Neural Network	2.4.4
MLP	Multilayer Perceptron	2.3.5
AESRD	Advanced Encryption Standard - Random Delay	2.6.2

List of Figures

List of Tables