SIDE-CHANNEL ANALYSIS WITH GRAPH NEURAL NETWORKS

SIDE-CHANNEL ANALYSIS WITH GRAPH NEURAL NETWORKS

Thesis

to obtain the degree of Master in Computer Science at Delft University of Technology, to be publicly defended on Thursday, April 29nd 2021 at 14:00

by

Vasco DE BRUIJN
Born in Rotterdam, The Netherlands.

Cyber Security Group,
Faculty of Electrical Engineering, Mathematics and Computer Science,
Delft University of Technology,
Delft, The Netherlands.

Thesis committee:

Chair	Prof.dr.ir. Inald Lagendijk , Faculty EEMCS, TU Delft
Daily Supervisors:	Dr. Elvin Isufi, Faculty EEMCS, TU Delft
	Dr. Stjepan Picek , Faculty EEMCS, TU Delft
Committee member:	Dr. Riccardo Taormina, Faculty CEG , TU Delft

Keywords: cyber security; graph neural networks; graph signal processing; profiled side-channel analysis

Contents

SUMMARY

In cyber security, side-channel attacks (SCA) are of interest because they target the vulnerabilities in implementation rather than inherent vulnerabilities in the algorithm. Profiled SCA is especially interesting as it assumes that the adversary has unlimited access to a clone device that can generate sufficient traces to create a profile of the device. The latest techniques used for profiled SCA are based on convolutional neural networks (CNN). However, CNN's are limited in scope in how they define convolution. By running the convolution over a graph instead, we can achieve a more flexible convolution method. Therefore, we want to apply graph neural networks (GNN) to SCA. To achieve this, we need to translate our SCA problem to a graph signal processing (GSP) problem. This is done by generating a graph based on the power traces on which the traces can be run as graph signals. Subsequently, this graph is used in a GNN to solve the GSP problem. We experiment with different GNN architectures to see how they differ in performance compared to SCA state-of-the-art. We also want to observe how our model deals with the different leakage models and if there is a considerable performance gap between them. We also want to see how GNNs deal with countermeasures such as masking and desynchronization. Finally, we perform hyper-parameter analysis to know whether we can reduce the number of learnable parameters without substantially decreasing the performance of our model. The numerical results demonstrate that our model is not competitive compared to state-of-the-art methods. The performance of our method is mainly derived from the classification multilayer perceptron instead of the graph convolutional filter layers. However, the results suggest that the graph convolutional filter layers are potentially helpful in existing SCA architecture as an initial layer that performs feature extraction.

1

INTRODUCTION

In cyber security, side-channel attacks (SCA) are of interest because they target the vulnerabilities in the implementation of the cryptographic algorithm rather than the vulnerabilities in the algorithm itself [43]. SCA exploits information leaked by the hardware implementation of cryptography, such as a timing attack [2], which exploits the difference in response time to determine the quality of the guessed password. In other words, if one guesses the first byte of the password correctly, the response time is higher than if one guesses the first byte of the password incorrectly [22].
For this research, we measure either the power usage or electromagnetic radiation of a chip during an encryption operation as the leaked information. The rationale behind these methods is that flipping the value of a bit takes more power than keeping the bit in the same state. So a technique based on these measurements assumes that hidden information can be derived from the number of bitflips in each operation. The encryption algorithm we will attack is the Advanced Encryption Standard (AES) method, also known as the Rijndael cipher [8] where the targeted hidden information is the (partial) key used during the encryption. One of the components in AES uses an exclusive OR (XOR) operation with the partial key as one of the inputs. Since an XOR operation is equivalent to a bitflip, this operation's output will be used as a leakage model of our attack. The leakage model describes how the desired secret information is leaked from the observed data.

We use different datasets [1], [37] which contain recorded leakage traces of a chip running an AES implementation with corresponding partial keys. A trace refers to a set of measurements taken across a cryptographic operation.
Profiled SCA (PSCA) is a category of SCA where the adversary is assumed to have access to a clone of the target device. For this reason, PSCA is deemed the most potent SCA technique since the adversary has access to everything but the key used for encryption. The template attack is a category of PSCA where the traces of the clone device are used to create a probability distribution for the states of the target device, which can be used to attack the target device [5]. It is deemed one of the most powerful techniques from a theoretical point of view. It generates a distribution that models the relationship between the leaked information and the device state. The main disadvantage of this technique

is that it requires many traces before an accurate distribution can be constructed. This problem can be mitigated by using machine learning techniques to derive the hidden information instead of using a probability distribution for a lower amount of traces to get a similar result[15].
As with most other machine learning applications, deep learning has emerged as a more powerful alternative for conventional machine learning within the domain of SCA[18]. The current state-of-the-art [51] used convolutional neural networks (CNN) as the base architecture for their deep learning modules. CNN's are powerful since they combine neighboring features into more abstract/higher-level features. Since the used features for this problem are generally time series, the adjacent features are naturally the ones that are close in the temporal space. However, when considering countermeasures such as random delays, and feature reduction by selection, this becomes a weaker designation of neighbor. Instead, another way of determining neighbors is desirable, requiring a method to encode 'arbitrary' neighbors.
One possible solution would be to use a Graph Neural Network (GNN) as a base architecture for deep learning. A GNN is a type of neural network which can process data represented in a graph domain. As the data is extended unto a graph, it is possible to define neighbors based on this graph instead of the temporal ordering of features. This introduces more flexibility when describing neighborhoods in a convolutional network depending on how the graph is designed. Another advantage of using a GNN is that it can exploit some symmetries in the traces, and this could help to reduce the number of parameters without impacting performance.
The goal of the SCA problem is to retrieve the (partial) keys used in the AES implementations using the power traces as leaked data. So the purpose of this thesis is to apply a GCNN to solve the problem above.

RQ1. How can the SCA problem be rephrased into a graph machine learning problem and which methods can we use to generate a graph based on a set of traces?

To address this research question, we propose to convert the SCA problem into a graph signal classification (GSC) problem. The goal of GSC is to label graph signals, so we transform the traces into graph signals. We propose to generate the graph by letting the time instances of the traces correspond to the nodes. The edges will be generated based on the correlation of the measurements at the different time instances.

RQ2. How does a graph neural network compare to state-of-the-art machine learning SCA techniques?

(a) How can we apply a graph neural network to reduce the number of learnable parameters without degrading performance?

(b) What is the influence of countermeasures on graph neural networks performance?

(c) How does a graph neural network perform on different leakage models?

(d) Which graph neural network architectures are (most) suitable to solve the graph machine learning problem?

To address these research questions, we analyze the performance of our graph neural network using different scenarios. We use hyperparameter tuning to find the best parameters for each architecture and compare the results. We also use datasets that each are generated using different countermeasure setups. Finally, we run different scenario's with both the Hamming Weight and the intermediate value as leakage model.

The answers to these research questions result in the following contributions:

1. We introduce a method to generate a graph based on a set of traces and how to use graph neural networks to solve side-channel analysis problems. The technique used to generate the graph is adaptable as it works regardless of the number of traces in the dataset or feature reduction.

2. We show that graph neural networks can solve the SCA problem by getting the guessing entropy sufficiently low to guess the key for the DPAv4 and ASCAD datasets. However, it could not achieve comparable results to state-of-the-art machine learning SCA techniques, especially concerning consistency.

The remainder of this document is organized as follows. Chapter 2 introduces the background required the problems . Chapter 3 discusses the literature related to this thesis. Chapter 4 contains the approach for translating this problem to a GNN problem. Chapter 5 describes the planned experiments and in Chapter 6 we conclude upon our thesis.

2

BACKGROUND

In this Chapter, we provide the background required to understand this thesis. In Section 2.1 we introduce the notation used in this report. In Section 2.2 we describe AES, the cryptography algorithm we plan to target. In Section 2.3 we describe the techniques used in side-channel analysis (SCA): we explain what traces and leakage models are. In Section 2.4 we discuss the theory required to understand graph neural networks (GNN). Furthermore, we compare different GNN architectures that are candidates to use in side-channel analysis.

2.1. NOTATION

Let bold upper-case letters such as $\mathbf{X}$ represent a matrix and let bold lower-case letters $\mathbf{x}$ represent a row vector while $\mathbf{x}^T$ represents a column vector (or rather, a transposed row vector). Furthermore, let non-bold letters such as X or x represent scalars or singular values. Let calligraphic letters such as $\mathcal{X}$ denote sets where $|\mathcal{X}|$ represents the size of a set and let x or $\mathbf{x}$ represent an arbitrary object drawn from that graph.
When we consider a graph, let $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ be a graph. consisting of the set of vertices $\mathcal{V} = [1, \ldots, |\mathcal{V}|]$ and the set of edges $\mathcal{E} = \{(i, j), \ldots\}$ where $i, j \in \mathcal{V}$.

2.1.1. TRACES

Let column vector $\mathbf{x}_i = [x_{i1}, \ldots, x_{iQ}]^\top$ represent the ith trace in the dataset as a feature vector, where x_{ij} corresponds with the measurement of trace i at time instance j. Let $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_P]$ be a $Q \times P$ matrix which corresponds to all traces in a given dataset, where P represents the number of traces and Q the measurements in each trace. Put differently, we can consider P as the number of items in the dataset and Q as the number of features, or time instances when traces are treated as time series. Let $\mathbf{x}^j = [x_{1j}, \ldots, x_{Pj}]^\top$ represent all measurements taken at time instance j . Let the vector containing the (partial) keys for the traces in the dataset be $\mathbf{k} = [k_1, \ldots, k_P]^\top$ where scalar k_i is the key associated with the ith trace.

When using AES, the used key is either 128, 192, or 256 bytes. It is customary to attack a single byte of the key at a time to reduce the complexity of this problem [15]. Therefore we use k_i to represent a single key byte from now on. Furthermore, t_i represents the plaintext byte associated with the ith trace. A byte consists of 8 bits which each can have as value either one or zero, so mathematically we define byte b as $b \in [0,1]^8$. However, we mostly use the decimal representation of the byte as that is easier to read. Let $K = |\mathcal{K}|$ represents the number of possible keys, where for a given K the set of possible keys is represented by $\mathcal{K} = \{0,\ldots,K-1\}$ When considering a single byte, the number of possible keys is $K = 2^8 = 256$.

2.2. AES

Advanced Encryption Standard (AES) is an encryption method used for many applications, which makes it an attractive target for cryptoanalysis [8]. AES-128 and AES-256 are implementations of AES where 128 and 256 refer to the size (in bits) of the key used in the implementation. We define encryption as $enc(t,k) = c$ where t is the plaintext we want to encrypt, k as the key used to encrypt, and c as the ciphertext resulting from the encryption. If we want to decrypt a ciphertext, we use a decryption function $dec(c,k) = t$ which returns the plaintext t using the ciphertext c and the key k as input. AES is a symmetric encryption method, so the key used for encryption is the same as the key used for decryption.
The encryption method used in AES works with multiple rounds, where each round uses a different round key. The number of rounds is dependent on the size of the key. At initialization, the plaintext is stored as an array of bytes called the state on which the AES operations are performed. The final state is directly output as ciphertext c.

1. `KeyExpansion` - Round keys are derived from key k using the AES key schedule. Each round requires a round key, plus one additional key for the initialization round
2. Initial round key addition
   (a) `AddRoundKey` – each byte of the state is combined with a byte of the round key using a bitwise exclusive or (xor) operation.
3. Rounds of transformation operations
   (a) `SubBytes` - each byte in the state is substituted according to the S-box lookup table
   (b) `ShiftRows` - the last three rows of the state are shifted cyclically
   (c) `MixColumns` - a linear mixing operation combining the four bytes in each column of the state
   (d) `AddRoundKey`
4. Final round
   (a) `SubBytes`

(b) `ShiftRows`

(c) `AddRoundKey`

For our thesis, the `SubBytes` step is the step of interest as we intend to focus our attack on the substitution operation. It should be noted that this attack only targets a single round and yields the round key as a result. However, the round keys used during encryption are the same as the round keys used for decryption, albeit in reversed order. Therefore obtaining the round keys is sufficient to break AES. From this point onward, we mean the round key as target when talking about keys.

2.3. PROFILED SIDE-CHANNEL ANALYSIS

Side-channel attacks (SCA) target the vulnerabilities in the implementation of the algorithm rather than inherent flaws in the algorithm [43] by exploiting information leaked by the implementation of the cryptographic algorithm. In this thesis, we look specifically at the information leaked by a chip performing a cryptographic operation. Profiled side-channel analysis (PSCA) is the category of side-channel analysis where the adversary is assumed to have full knowledge of the used cryptographic algorithm as well as a copy of the device which they are free to program. The goal of PSCA is to build a "profile" of the device running the cryptographic algorithm, which can be used to accurately predict the secret key based on one (or multiple) traces of leaked information. An example of a trace can be seen in Figure 2.1.

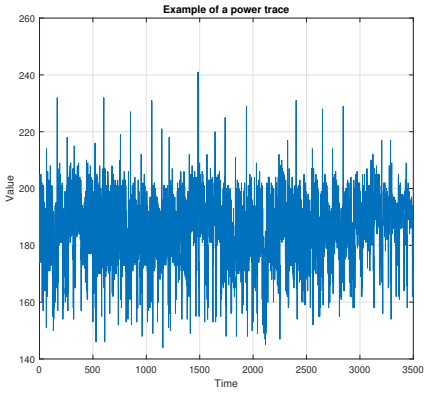

Figure 2.1: Example of a trace represented as a time series. Each sample per time unit corresponds with the measurement taken of the power usage of the chip at that time.

In our scenario, the attacker has obtained a set of profiling traces T_p from a device in which key k^* is known and used to build the leakage profile in the profiling phase. In the attacking step, the attacker obtains additional traces T_a from the device to determine the secret key k^*. In this thesis, we consider the goal to learn from the traces to break the key using graph neural networks.

2.3.1. LEAKAGE MODEL

While key k^* is the information we are interested in retrieving, it is not directly related (in a mathematical sense) to the values of the traces. Therefore, the key needs to be mapped to an intermediate value (IV) related to the measured leakage. A leakage model maps the intermediate values to the leakage measurements.

For this mapping, we select an operation of the cryptographic algorithm to attack. As we assume that we know the used plaintext t, we chose an operation that uses both the key and plaintext as input value [15],[27]. To reduce the complexity of the attack, we only attack a single byte of the key at a time. For AES, the operation which is attacked is the Sbox-substitution operation. The mapping y maps plaintext byte t and key byte k^* to an intermediate value assumed to be related to the corresponding traces x. This mapping is noted as

$$y(t, k^*) = \text{Sbox}[t \oplus k^*], \tag{2.1}$$

where Sbox[.] is the substitution look-up table used in AES and $\oplus$ is the bitwise exclusive-OR operator. The Sbox maps a single byte input to a single byte output, so y will be considered a byte. Each entry in the Sbox is unique by design which means that the Sbox operation can be treated as a bijective mapping. I.e. $\text{Sbox}[b_0] \neq \text{Sbox}[b_1]$ if $b_0 \neq b_1$ and $\text{Sbox}[b_0] = \text{Sbox}[b_1]$ if $b_0 = b_1$. The precise definition of the Sbox is not relevant for our thesis and can be found in [8].
Notice that even when k^* is a fixed key (i.e. the value of k^* is the same for each trace), y can still take different values as it is also dependant on the used plaintext p. For example, let $k^* = 33$ and $t_0 = 32, t_1 = 3$. This gives $k^* \oplus t_0 = 1, k^* \oplus t_1 = 34$ as intermediate result and $y_0 = \text{Sbox}[k^* \oplus t_0] = 124, y_1 = \text{Sbox}[k^* \oplus t_1] = 221$ as final result. This small example shows that the intermediate values can differ even if the same key is used to calculate them. This allows us to treat this problem as a classification problem without having to deal with the issue that the training data is biased towards a single label.

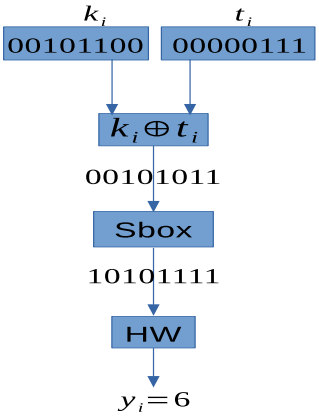

Figure 2.2: Example of calculating the leakage model. The used input is $k_i = 44, t_i = 5$ and each step of the calculation shows the result as bitstring.

It is known that devices tend to have leakage which can be correlated to the Hamming Weight of the intermediate value [27],[36],[30],[15],[25]. The Hamming Weight (HW) represents the number of ones in the bit-string representation of the input. For example, $HW(5) = 2$ as the byte representation of 5 is 00000101 which contains two ones. Similarly, $HW(72) = 2$ as well since the byte representation of 72 is 01001000, comprising two ones. So when using the Hamming Weight as leakage model, the mapping y can be written as:

$$y(t, k^*) = HW(\text{Sbox}[t \oplus k^*]) \tag{2.2}$$

When considering the Hamming Weights of a single byte, the possible values range from zero (bitstring with only zeros) to eight (bitstring with only ones). This means the Hamming Weight can reduce the label space from 256 to 9 different labels/classes. This reduction of the number of classes is a valuable incidental property as it is more difficult for ML to discriminate between many possible classes [29]. However, this has a disadvantage as well. Not each Hamming Weight corresponds to the same number of intermediate values. For example, $HW = 0$ corresponds to one possible value, while $HW = 1$ corresponds to eight possible values. So when using the HW leakage model, we incur class imbalance which might harm our performance [32].

2.3.2. GUESSING ENTROPY

Unlike common classification problems, side-channel analysis uses different metrics to measure the performance. In general, the output of a classification problem would be a probability vector $\mathbf{p} \in [0,1]^C$ where C represents the number of possible classes. The entry p_c represents the confidence that the classified object has class c as a label. With normal classification problems, we are only interested in the predicted class of the object, which corresponds with the class c with the highest confidence p_c. On the other hand, with SCA we are interested in guessing the secret key. This is done under the assumption that we can verify whether our guess is correct, but it is unfeasible to find the key by randomly trying each guess. Therefore, we are not just interested in whether the key guess with the highest confidence is correct, but rather how many guesses it takes to find the right key. The metric we use to measure the number of key guesses is called the Guessing Entropy (GE), which is used as standard in literature [41],[44]. In the remainder of this subsection, we shall describe how the GE is calculated and how we use it.
Let guessing vector $\mathbf{g} = [k_0, ..., k_K]^\top$ where $k_i \in [0, ..., K-1]$ represent the output of some side-channel attack on the secret key k^*. Vector $\mathbf{g}$ represents a guessing vector which contains all possible keys ordered by decreasing the probability of being the correct prediction for k^*; so k_0 is the most likely candidate and k_K is the least likely candidate.
To calculate the GE, let us first define the Key Rank (KR) which we need to calculate to obtain the GE. The KR represents the position of secret key k^* in a guessing vector $\mathbf{g}$,

$$\text{KR}(\mathbf{g}, k^*) = j : k_j \in \mathbf{g} \wedge k_j == k^* \tag{2.3}$$

where k_j represents the key guess with position j in the guessing vector $\mathbf{g}$. The guessing entropy represents the average position j of k^* in $\mathbf{g}_i$ for a set of guessing vectors $\mathbf{G} = [\mathbf{g}_0, \ldots, \mathbf{g}_n]$. In other words, the guessing entropy is the mean of the key rank applied to multiple guessing vectors, so we can use the definition of KR as in Equation (2.3) to

calculate the GE.

$$\mathrm{GE}(\mathbf{G}, k^*) = \frac{1}{|\mathbf{G}|} \sum_{\mathbf{g}_i \in \mathbf{G}} \mathrm{KR}(k^*, \mathbf{g}_i) \tag{2.4}$$

A low GE indicates the secret key corresponds with one of the likelier guesses in the guessing vectors, while a higher value indicates the secret key corresponds with worse guesses.

It is common to combine the results of multiple traces corresponding to the same key . Let the output of an attack on the secret key k^* for trace i be $\mathbf{p}_i = [p_{i0}, ..., p_{ij}]^\top \in [0,1]^K$ where p_{ij} represents the probability that the candidate key k_j is equal to the secret key, i.e. $k_j = k^*$. If the number of traces used for the attack is n, then the combined key guessing vector $\mathbf{c} = [c_0, ..., c_{K-1}]^\top$ can be calculated using the log-likelihood principle for each entry c_k

$$c_k = \sum_{i=1}^{n} \log(p_{ki}) \tag{2.5}$$

where p_{ki} is the estimated probability for candidate key k using trace i. In other words, c_k contains the likelihood that candidate key k is equal to target key k^* where the likelihood is based on the combination of the guessing vectors resulting from the traces. We show the process of calculating the GE using a simple example in Figure 2.3

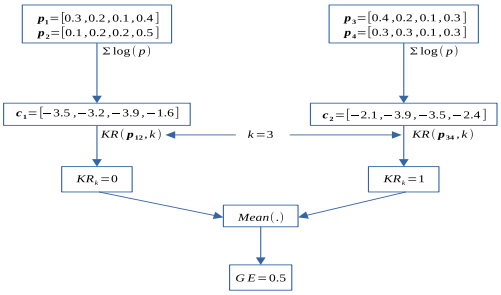

Figure 2.3: An illustration of the calculation of Guessing Entropy. To simplify the explanation, we consider only four different keys. We define a set of classification probability vectors $\{p_1, p_2, p_3, p_4\}$. We first combine the pairs as in Equation (2.5). We use the resulting vector to calculate the Key Rank as per Equation (2.3). Finally, we take the mean of the Key Ranks to retrieve the Guessing Entropy as per Equation (2.4).

The output of an attack will result in a classification vector $\mathbf{p} \in [0,1]^C$, where C is the number of classes which depends on the used leakage model. Therefore, we need to map $\mathbf{p}$ to a probability vector $\mathbf{p}' \in [0,1]^K$. We do this by taking the operation to calculate the leakage model as described in Equation (5.1) and applying it to each candidate key $k \in \mathcal{K}$:

$$p'_k = p_y : y = y(t, k) \tag{2.6}$$

Where t is the corresponding plaintext. Since the number of classes when using the hamming weight as leakage model is nine, and the number of possible keys is 256, p' will contain a lot of duplicate probabilities. This is solved by combining the results as described in Equation (2.5). This works since different plaintexts t_1, t_2 correspond (in general) with different Hamming Weights $y(t_1, k^*), y(t_2, k^*)$. So the set of keys corresponding to the same HW is different for each plaintext. Therefore, when combining multiple outputs, we get a unique probability for each key, allowing us to calculate the guessing entropy properly.

2.4. GRAPH NEURAL NETWORKS

A Graph Neural Network (GNN) is a type of neural network used for graph-related deep learning. GNN's work by running the input feature vectors on graphs. The main goal of GNNs for our purpose is to solve different classification tasks: they can be used to classify nodes or entire graphs [38]. They can also be used for other tasks like link prediction [52] or regression tasks [17], but those applications are beyond the scope of this thesis. The classification task that is the focus of this thesis is graph signal classification. Graph signal classification seeks to find a label for a signal (or feature vector) run on a graph.

2.4.1. SIGNAL GRAPHS

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ represent a graph with vertices $\mathcal{V} = \{1, ..., N\}$ and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. An example of a simple graph can be found in Figure 2.4

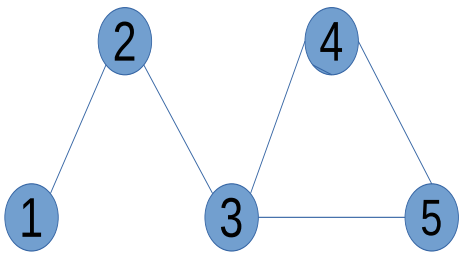

Figure 2.4: An illustration of a undirected graph with 5 vertices.

Consider feature vector $\mathbf{x} = [x_0, \ldots, x_N]$ where the number of features equals the number of vertices $|\mathbf{x}| = |\mathcal{V}|$. Each feature x_v in $\mathbf{x}$ is assigned to corresponding vertex $v \in \mathcal{V}$. In which case $\mathbf{x}$ can be considered a graph signal defined on the vertices of graph $\mathcal{G}$. A graph signal classification problem requires a classifier that, given a graph $\mathcal{G}$ and a graph signal $\mathbf{x}$ as input, outputs a label prediction y for the graph signal. We give a visual example of how a graph signal works in Figure 2.5.

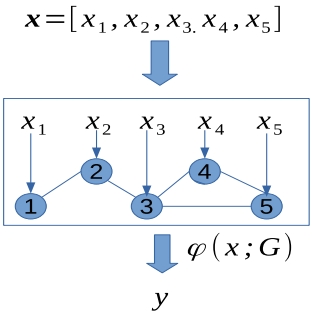

Figure 2.5: A illustration of a graph signal. The graph signal $\mathbf{x}$ is mapped on the graph $\mathcal{G}$. The goal of the graph signal classification is to find some function $\phi(\mathbf{x}, \mathcal{G}) = y$ which is able to find the correct label y for the graph signal

2.4.2. SHIFT OPERATOR

There are multiple notations to represent the edges of a graph. In graph processing theory, it is common to represent the edges as a shift operator matrix $\mathbf{S} \in \mathbb{R}^{N \times N}$. The shift operator is defined such that entry $s_{ij} \neq 0$ if $(i, j) \in \mathcal{E}$. The simplest shift operator is the adjacency matrix $\mathbf{A}$ where $s_{ij} = A_{ij} > 0$ and A_{ij} is some edge weight corresponding to edge (i, j). If the edges of the graph are not weighed, each edge $(i, j) \in \mathcal{E}$ corresponds with entry $A_{ij} = 1$. Another commonly used shift operator for undirected graphs is the graph Laplacian matrix $\mathbf{L}$. Define the diagonal degree matrix $\mathbf{D} \in \mathbb{R}^{N \times N}$ such that $D_{ii} = \sum_j A_{ij}$, then the Laplacian can be defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$. The Laplacian matrix is essential for spectral graph analysis which we touch upon briefly in Section 3.3.

2.4.3. GRAPH CONVOLUTIONAL FILTER

A graph filter is a function that takes a graph signal as input, processes it over graph $\mathcal{G}$ and returns a modified graph signal as output. A graph convolution is a type of graph filter that operates on graph shift operator $\mathbf{S}$ and is defined as

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{K} h_k \mathbf{S}^k \tag{2.7}$$

where $\mathbf{h} = [h_0, \ldots, h_K]$ are the filter coefficients. The purpose of this filter is to simulate the signal traveling through the graph and propagating through its neighbourhood (i.e. convolving) for K hops, where a filter simulating K hops is said to be of order K. By applying a graph convolutional filter to a graph signal, we get the output:

$$\mathbf{x}' = \sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x} \tag{2.8}$$

where $\mathbf{x}$ represents the input graph signal and $\mathbf{x}'$ represents the output graph signal. Figure 2.6 shows how a signal propagates through a graph. Figure 2.7 gives a graphical overview of how the different components of the convolutional filter interact which each other.

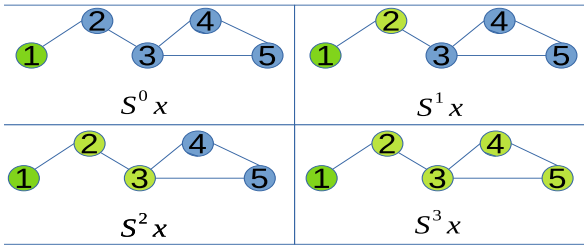

Figure 2.6: An illustration of a signal, origination from node 1, propagating thorough the graph. The dark green node represents the signal and the light green nodes represents the propagated signal. S represents the shift operator

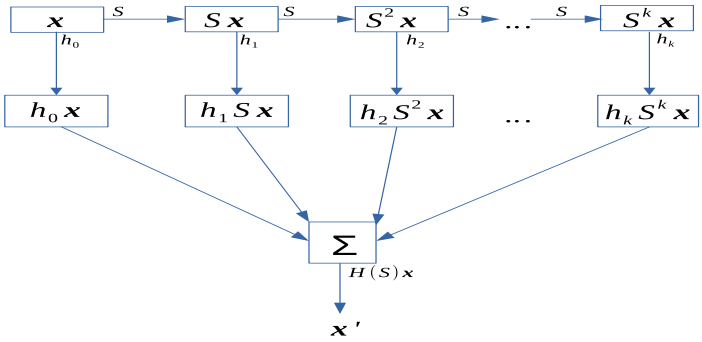

Figure 2.7: Illustration of how the graph convolution filter works. Each transition to the right in the illustration represents a step taken in the propagation of the signal.

2.4.4. GRAPH NEURAL NETWORK ARCHITECTURES

We consider a GNN as an architecture consisting of L layers where each layer l takes $\mathbf{x}_{l-1}$ as input, and outputs $\mathbf{x}_l$. The initial input $\mathbf{x}_0 = \mathbf{x}$ is the graph signal used as input to the GNN. Each layer consists of two parts: the graph filter $\mathbf{H}_l(\mathbf{S})$ used and a nonlinearity function $\sigma(.)$. By combining these parts, we get the following propagation rule:

$$\mathbf{x}_l = \sigma(\mathbf{H}_l(\mathbf{S})\mathbf{x}_{l-1}) \tag{2.9}$$

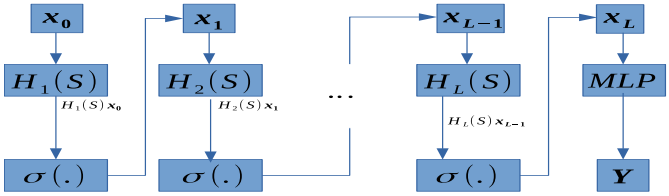

Figure 2.8: Illustration of how a graph neural network works using a generic graph filter $\mathbf{H}_l(\mathbf{S})$. The GNN uses $\mathbf{x}_0$ as input and outputs $\mathbf{Y}$. Each column of operations represents a layer in the GNN.

For now, we assumed that a graph signal $\mathbf{x}$ consists of a single node feature. However, a graph signal can consist of multiple node features. In that case, it is denoted as matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$ where F represents the number of features. Therefore, we change the notation from the propagation rule introduced in Equation (2.9) to

$$\mathbf{X}_l = \sigma(\mathbf{H}_l(\mathbf{S})\mathbf{X}_{l-1}) \tag{2.10}$$

As the purpose of our GNN is signal classification, we require a function which maps the output signal $\mathbf{X}_l$ to a classification vector $\mathbf{p} \in [0,1]^C$. We use a multilayer perceptron (MLP) consisting of fully connected layers to perform this mapping. We define a layer of the MLP as

$$\mathbf{x}_l = \sigma(\mathbf{\Theta}_l \mathbf{x}_{l-1}) \tag{2.11}$$

where $\mathbf{\Theta}_l = \mathbb{R}^{F_{l-1} \times F_l}$ represents the learnable weights of the fully connected layer. F_l denotes the size of vector $\mathbf{x}_l$. Note that the input of the MLP is a vector, so when we use matrix $\mathbf{X}_l$ as input, we first need to flatten the matrix into a vector.
Depending on the type of graph filter we choose for all the layers, we get a different kind of GNN architecture. In the following subsections, we discuss three different architectures on which we have decided to focus.

GRAPH CONVOLUTIONAL NEURAL NETWORK

The graph convolutional neural network (GCNN) uses the graph convolutional filter introduced in Section 2.4.3 as graph filter. By adapting the graph convolutional filter as defined in Equation (2.8) we define a layer in a GCNN as

$$\mathbf{X}_l = \sigma\Big(\sum_{k=0}^{K} \mathbf{H}_{lk}\mathbf{S}^k\mathbf{X}_{l-1}\Big) \tag{2.12}$$

.

Here, $\mathbf{X}_l$ represents the output of layer $l = 1,\ldots,L$, where the initial features $\mathbf{X}_0$ are equal to the input $\mathbf{X}$ of the GCNN ; $\mathbf{S} \in N \times N$ represents the shift operator, K is the num-

ber of propagation steps taken within a layer. F_l represents the number of features output by each layer. $\sigma(\cdot)$ the non-linearity function (e.g. ReLU) and $\mathbf{H}_{lk} \in \mathbb{R}^{F_{l-1} \times F_l}$ represents the filter bank at layer l for propagation step k. The dimensions of $\mathbf{H}_{lk}$ corresponds with respectively the number of input features for layer l and the number of output features for layer l. For simplicity, we decided that each layer has the same number of output features F so for each layer $F_l = F$. This convention is also adapted for the other architectures.
The main strength of the GCNN is that it convolves the neighboring input features according to the input graph. A regular CNN convolves neighboring input features based on the ordering of the features, assuming that features that are close according to this ordering are considered neighbors. In Section 3.2 we explain that state-of-the-art(sota) uses CNN. So we know that convolving the features can yield good results. We could simulate a CNN with a GCNN by running the GCNN on a cyclic graph [1]. Therefore, if we can find a graph that is a good representation of the neighborhood, we should get good results. In short, since the GCNN is conceptually similar to CNN and CNN has an excellent track record for this subject, we think that the GCNN is a good candidate architecture.

GRAPH CONVOLUTION ATTENTION NETWORK

The graph convolution attention (GCAT) filter builds upon the convolution filter by adding an attention mechanism [16]. The purpose of an attention mechanism is to learn weights assigned to edges. Those weights are used to find and reduce the impact of low-quality edges.
The GCAT filter uses $\mathbf{\Phi} \in N \times N$ as an attention matrix. Matrix $\mathbf{\Phi}$ shares the sparsity pattern of $\mathbf{S}$ so it acts like a graph filter as well.

$$\mathbf{X}_l = \sigma\Big(\sum_{k=0}^{K} \mathbf{H}_{lk} \mathbf{\Phi}_l^k \mathbf{X}_{l-1}\Big) \tag{2.13}$$

The matrix $\mathbf{\Phi}_l$ is learned from the output of the previous layer $\mathbf{X}_{l-1}$. Each layer uses a different matrix $\mathbf{\Phi}_l$, but we omit layer index l to simplify the notation. The entries of $\mathbf{\Phi}$ are calculated using a score a_{ij} :

$$\alpha_{ij} = \sigma(\mathbf{e}^T \left[[\mathbf{X}_{l-1}\mathbf{B}]_i, [\mathbf{X}_{l-1}\mathbf{B}]_j \right]^T) \tag{2.14}$$

where $\mathbf{B} \in \mathbb{R}^{F \times F}$ and $\mathbf{e} \in \mathbb{R}^{2F}$ are both learnable parameter matrices. In Equation (2.14) we take the feature matrix $\mathbf{X}_{l-1}$ and mix them with the coefficients in $\mathbf{B}$. This mixture results in graph signal matrix $\mathbf{X}_{l-1}\mathbf{B}$ where each node i has F features which correspond to row vector $[\mathbf{X}_{l-1}\mathbf{B}]$. We concatenate the row features of nodes i and j which is multiplied with vector $\mathbf{e}$.

[1] A cyclic graph is a graph where each node n is connected with its direct successor $n+1$

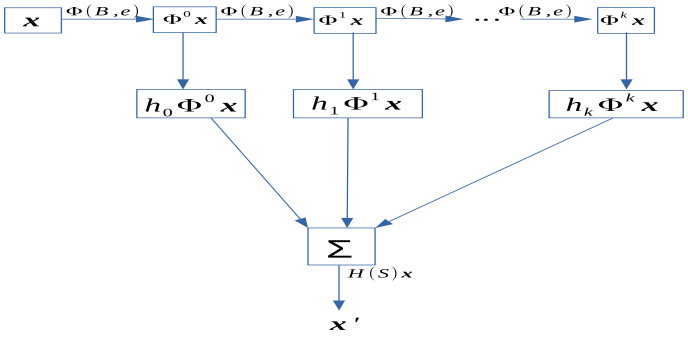

Figure 2.9: Illustration of how the graph convolution attention filter works. Note that the filter is similar to the convolutional graph filter as this illustration does not show how $\mathbf{\Phi}$ is learned or that GCAT uses different shift operators for each layer.

It is possible to use α_{ij} directly as the entry for Φ_{ij}, but the model in [16] utilizes additional attention sparsity. This means that within a neighborhood $\mathcal{N}_i$ of node i, we want to boost the score of neighbors with a high attention score and dampen the score of neighbors with a low attention score. We use a local softmax operator (SoftMax) to achieve this. After applying SoftMax, the highest score in the neighborhood will approach 1 and the other values will approach 0. This effect will diminish if the scores in the neighborhood are relatively close to each other. The local SoftMax function is defined as:

$$\Phi_{ij} = \exp(\alpha_{ij}) \times \left(\sum_{j' \in \mathcal{N}_i \cup i} \exp(\alpha_{ij'}) \right)^{-1} \tag{2.15}$$

In addition to encouraging attention sparsity, SoftMax also adds normalization as it ensures that each value Φ_{ij} is between zero and one. Furthermore, the sum of all values in a neighborhood is guaranteed to be one.

$$\sum_{j' \in \mathcal{N}_i \cup i} \Phi_{ij'} = 1 \tag{2.16}$$

GCAT is similar to GCNN as both use a convolutional graph filter as their main component. However, GCNN uses a fixed shift operator $\mathbf{S}$ in its graph filter to represent the edges in the graph. Conversely, GCAT does not use a fixed shift operator but utilizes a learnable shift operator $\mathbf{\Phi}$ to represent the graph. This means that GCAT can correct mistakes in the edge/neighborhood generation step by making the shift operator a learnable component and having a lower weight on the 'erroneous' edges. In more general terms, GCAT has a higher number of learnable parameters than GCNN, as shown in Table 2.1. This implies that GCAT can learn more complex representations compared to GCNN, which makes it more expressive. However, this may be a disadvantage if the representation turns out to be relatively simple. In that case, a more complex model will likely suffer from overfitting.

Furthermore, the attention mechanism in GCAT might turn out too strict and penalize too many valuable neighbors. This risk is enhanced by the SoftMax operator which forces attention sparsity. In other terms, the GCAT forces one neighbor for each node, even if it would be more logical to have multiple neighbors. Therefore, a GCAT model might be unsuitable for graphs in which nodes depend on multiple edges that are individually weak.

To conclude, whether the GCAT model is a suitable architecture depends heavily on the graph on which the architecture is applied. On the one hand, GCAT is helpful to correct faulty edges in our graph generation method. On the other hand, it is detrimental when there is a small difference between the importance of each edge.

EDGE VARYING GRAPH NEURAL NETWORK

The edge varying graph neural network (EdgeNet) is a GNN which uses an edge varying filter as graph filter [16]. Let $\mathbf{\Phi}^{(0)}$ be the identity matrix $\mathbf{I}_N$ and let $\mathbf{\Phi}^{(1)}, \dots, \mathbf{\Phi}^{(K)}$ be K matrices with the same sparsity pattern as $I_N + \mathbf{S}$. From here on, we can consider $\mathbf{\Phi}^{(k)}$ to be a shift operator as well. Let us also define product matrix $\mathbf{\Phi}^{(k:0)} = \prod_{k'=0}^{k} \mathbf{\Phi}^{(k')}$. Notice that this product matrix looks similar to the convolutional graph filter, with as main difference that the convolutional graph filter uses a fixed shift operator for each shift while $\mathbf{\Phi}^{(k:0)}$ uses different shift operators or for each shift. Using this as our shift operator, we can define a layer of an EdgeNet as:

$$\mathbf{X}_l = \sigma\Big(\sum_{k=0}^{K} \mathbf{H}_{lk} \mathbf{\Phi}_l^{(k:0)} \mathbf{X}_{l-1}\Big) \tag{2.17}$$

Notice in Equation (2.17) that EdgeNet has to learn $K \times L$ different shift operators and compare this to GCAT, which only has to learn L different shift operators or GCNN which does not have to learn any shift operators. This means that the EdgeNet has more expressive power compared to the other methods. However, having to train the shift operators from scratch is expensive compared to the GCAT method, especially since the GCAT method has a lower number of parameters to calculate per shift operator ((F^2+F)) compared to EdgeNet $|\mathscr{E}|$ [2]. The EdgeNet has the advantage over the GCAT. It can correct for erroneous edges and is not explicitly inclined towards shifting all weight towards a single edge per neighborhood. We could also consider the GCNN and the GCAT as constrained variants of the EdgeNet [16]. In which case, assuming shared parameters, each solution found by the GCNN and GCAT can also (in principle) be seen by the EdgeNet. So theoretically, the EdgeNet is the most powerful architecture. In practice, the EdgeNet may not find this solution due to the large degree of freedom and that learning algorithms are not always guaranteed to find the most optimal solution.

[2] In general, the number of edges in a graph will be substantially higher than the number of features.

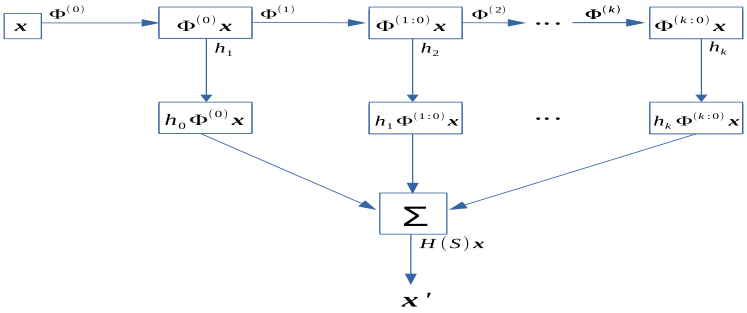

Figure 2.10: Illustration of how the edge varying graph filter works. Notice how each propagation step uses a different shift operator unlike the convolutional graph filter

To conclude which architecture is the most suitable for our problem is heavily dependent on the quality of our generated graph. If the graph contains many redundant edges, the GCAT architecture is the most suitable as it is inclined to reduce the number of edges. If the graph is of low quality in other areas, the EdgeNet may be the better solution as it has more freedom concerning the edge weights than the GCAT. On the other hand, if the graph represents the underlying structure, error correction would not add much to our model. In which case the GCNN is the best architecture as it assumes the given graph as ground truth and does not spend additional resources for fault correction.

Architecture	Parameters
GCNN	$L(K+1)F^2$
GCAT	$LR(F^2+2F+F^2(K+1))$
EdgeNet	$L(K(M+N)+N)F^2$

Table 2.1: Properties of different graph neural network architectures. Legend: L-number of layers; K-order of graph filter; F-number of features; N-number of nodes; M-number of edges; R-number of attention branches

2.5. CONCLUSION

This Section summarizes the content we have discussed in this Chapter. In Section 2.1 we introduced the notation used in this report. In Section 2.2 we explain the cryptographic algorithm we want to attack and which component we target. In Section 2.3 we explained the side-channel analysis problem we want to solve. First, we briefly described the traces and how we want to use them to obtain the secret key. Next, we explained the leakage model which allows us to extract the desired leakage from the traces. Finally, we present how to evaluate the results of a side-channel attack by using guessing entropy. In Section 2.4 we discuss graph neural networks and the necessary background. First, we introduce signal graphs, which are graphs on which a signal is run. Next, we discuss how we use shift operators as a representation of the edges in a graph. Subsequently, we introduce graph convolutional filters, which are used to filter graph signals and are

the main component of graph neural networks. Finally, we propose three different GNN architectures which we deem suitable to solve our problem and compare the different architectures.

3

Related Works

This chapter contains an overview of the literature relevant to this thesis. In Section 3.1 we briefly introduce power analysis methods which were used before profiled side-channel analysis. In Section 3.2, we describe the history behind profiled side-channel analysis and what the current state-of-the-art is. In Section 3.3, we show works about graph neural networks. In Section 3.4, we discuss some other applications of graph neural networks in the cybersecurity domain.

3.1. Non-Profiled Power Analysis

Power analysis refers to the use of the power usage of a device as the leveraged leaked information. The earliest used method used is Simple Power Analysis (SPA) [22]. SPA is a method that directly leverages power consumption traces by visually inspecting patterns in the traces. It works on the assumption that there no countermeasures are taken to protect against SCA. Therefore it was sufficiently powerful during the early days of cryptography. It is still possible to utilize SCA for specific purposes like identifying conditional branches or correlating values in modular multiplication and exponentiation. A more advanced method is the Differential Power Analysis (DPA) [20]. DPA is a statistical method used to find correlations in the data. It partitions the traces and finds the differences in the averages of these subsets. If the partition criterion is uncorrelated to the measurements in the traces, the differences in the subsets' averages will approach zero given a sufficiently high number of traces. If the partition criterion is correlated to the measurements, the averages will approach a non-zero value given a reasonably high number of traces [21].

3.2. Profiled Side-Channel Analysis

Profiled Side-Channel Analysis (PSCA) is a type of side-channel analysis that uses a model built from a set of traces separate from the set of traces used for the attack. In this section, we distinguish between three different categories of PSCA. The template attack uses a purely statistical method to create its profile [5]. It uses the profiling traces to develop a

multivariate Gaussian model of the noise in the traces. This model is used in the attacking phase as a prior to calculating the likelihood for the different keys.
The machine learning (ML) techniques treat the PSCA as a classification problem for machine learning. Multiple machine learning techniques have been used for PSCA, but the most common are Support Vector Machines (SVM) and Random Forests(RF).
Deep Learning (DL) techniques make use of neural networks to solve PSCA problems. DL techniques are technically speaking a subset of ML techniques, but due to the recent popularity of DL in PSCA[24],[3] [32][18] we consider it a separate category.

3.2.1. TEMPLATE ATTACK

The template attack works by using the device to derive a distribution of the noise generated by the device for each possible secret key. This distribution is used in tandem with the average signal for the given key to compute a template for each possible key. These templates are used to classify a sample by calculating the likelihood of generating the given sample for each template. The template attack (TA) works under the assumption that the adversary has unlimited access to the device it tries to compromise [5]. This allows the adversary to use an unlimited number of traces to support their attack. The adversary has complete control over which parameters (e.g., plaintext and secret key) are used to generate the traces. This allows the template attack to break implementations believed to be immune to earlier attacks such as DPA. This is because TA seeks to build a multivariate profile that has more explaining power compared to DPA, which seeks correlations.
A drawback of the template attack is that it requires one thousand samples per possible key to getting a good approach of the mean signal and the multivariate distribution of the noise according to [5]. Even though the adversary has a clone device available, recording many traces is an expensive endeavor.

3.2.2. COUNTERMEASURES

Countermeasures are used to make it more difficult for the adversary to get useful information out of the leakage. One could either make it more difficult (or even impossible) to capture the leakage or try to make it more challenging to extract useful information from the leakage. Since we are working from a PSCA perspective, we are only interested in the latter type of countermeasures.

With the advent of using SCA to break cryptographic algorithms, countermeasures were developed to shield against side-channel attacks like the template attack or DPA. Methods commonly used for countermeasures include hiding countermeasures and masking. Both approaches have shown to be effective countermeasures against template attacks [42], [46].
The basic idea behind masking is to randomize the secret variables (i.e., the secret key) during a cryptographic operation by applying a randomly generated mask [42]. In general, this can be explained as follows. When we input a plaintext, a mask is randomly generated. This mask is applied to the plaintext to create a masked variable. Subsequently, the encryption algorithm is applied to the masked variables. Simultaneously, the device calculates a de-masking variable that will return the original variable if applied to the masked variable.

Hiding countermeasure is a type of countermeasure which makes use of time randomization to improve protection against SCA [46]. There are multiple options for implementing de-synchronization: one method is to introduce random delays (jitter) between different operations[7]. Another method is to introduce randomness in the order of operations [13]. The technique used for the ASCAD dataset[1](which we plan to use) was to randomly shift the time instances, so it becomes more challenging to align points of interest. The main goal of this approach is to make the temporal relation between the cryptographic operation and the measured traces less straightforward.

3.2.3. MACHINE LEARNING IN SCA

One of the first approaches for using machine learning for PSCA was in [15] where the authors used support vector machines (SVM) to perform the classification task to compare it with the template attack. The results show that the SVM has comparable performance to the TA and outperforms the TA in some cases. The results also show that SVM requires fewer traces for a good performance, where only at 500 traces a significant decrease in quality was observed. SVM and Randoms Forest (RF) are shown to break countermeasures such as masking in [26]. This work shows that ML techniques like SVM and RF can outperform TA when countering countermeasures is concerned. This work also observed that the noise present in the measurements does not follow a Gaussian distribution. Recall that TA works under the assumption that a Gaussian distribution can model the noise. This explains why ML techniques can outperform TA, as (most) ML techniques do not make this assumption. SVM and RF have commonly been used in multiple other works as well [34][23][33][31][14].
From a learning point of view, PSCA has some particularities because leakage models are used as labels instead of the keys directly, which are addressed in the work of [32]. The first one is that when the Hamming Weight (HW) is used as leakage model, a class imbalance is incurred as not every HW is mapped to an equal number of intermediate values (i.e. a HW value of 0 is mapped to a single value while a value of 4 is mapped to 70 different values). Another issue is that guessing entropy as a metric is more complex than the usual ML metrics like accuracy. Both of these issues are addressed where the authors compare side-channel metrics to regular machine learning metrics and use balancing techniques such as SMOTE (generation of artificial data points for the imbalanced class based on intrapolation) to deal with the class imbalance problem. The work in [48] shows that guessing entropy can give wrong results as a metric as well and proposes an improved guessing entropy based metric which makes use of bootstrapping the test data instead of using all test data. This method does have the unfortunate caveat that it requires a larger test set for this method to work properly compared to the simple guessing entropy.

More recently, deep learning has emerged as a more effective alternative for machine learning in PCSA. In side-channel analysis, there are two deep learning methods that are commonly used: the multilayer perceptron (MLP) and convolutional neural networks (CNN). The use of MLP for PSCA was introduced in the work of [12]. It is interesting to note that this work uses two MLPs for their attacks. The first MLP counteracts the masking countermeasure by directly finding the mask. The found mask is subsequently used

by the second MLP which performs the key recovery. The subsequent works we mention in this thesis combine these functionalities in a single network. Which is to say, recovering the masks is done implicitly in the networks mentioned afterwards. The work in [24] is one of the first to use deep learning for SCA. The authors used MLP and CNN in their experiments (as well as the auto-encoder) as methods for a side-channel attack. This work shows that DL methods outperform both TA and earlier ML methods (SVM and RF). As opposes to SVM, (and RF) DL methods can exploit temporal relations in the traces. Also, CNN has the ability to generate high-level features from the raw data.

In [3] the authors continue with this line of thought and note that CNN's are robust to misalignment of data which is a helpful property as de-synchronization is a commonly used countermeasure against these attacks. CNN's can perform feature extraction on datasets without manual feature pre-processing. This is useful, as this removes the need for an extra pre-processing step in which the attacker needs to select points of interest. In the case of de-synchronization countermeasures, the attacker might also need to re-synchronize the traces. Both of these methods are heuristic-based which makes it likely that those methods are not optimal for preprocessing. Moving the feature processing step in the neural network and making it learnable makes it possible to optimize the feature extraction. Although it should be noted that manual feature processing can still be helpful for feature reduction to speed up learning.
Hyperparameter choice is an important aspect for the performance of any DL method, as the choice of the hyperparameters is linked to the complexity of the model. Complexity is defined in terms of the number of learnable parameters in a model. In general, it holds that a more complex model can solve more difficult tasks, at the expense of time and memory. A complex model has a large solution space, so for a simple problem, a complex model might find an over-complex solution while a simple solution might give better results. For example, a polynomial of order $O(x^1)$ can provide a better fit to a set of points on a straight line compared to a polynomial of order $O(x^9)$. To get more insight into the choice of hyperparameters, the work of [51] introduces a standard to compare different models concerning the chosen hyperparameters by comparing the complexity of the models. This work shows that it is possible to find more efficient models than other state-of-the-art while having a smaller complexity. It should be noted that with sufficient hyperparameter tuning on the same dataset, it is very likely to find a configuration that gives a good performance on that specific dataset. In fact, it would be a sign of robustness if the performance of a neural network is consistently good- not necessarily great- regardless of the choice of hyperparameters up to a certain degree. Such a neural network was introduced in [18]. In this work the authors propose a general CNN which has a good performance on multiple datasets as compared to the CNN's introduced in [3] or [1] which was tuned to a single dataset. The work does note that while the base architecture gives a good performance on all proposed datasets, each dataset does need some fine-tuning with regards to hyperparameters. This work also shows that adding Gaussian noise to the traces increases the performance of their model for most datasets. The addition of this noise is equivalent to adding a regularization term to the objective function. This causes the model to be more robust as the addition of noise means that the model learns multiple permutations of the input data, making it less likely to gener-

alize.

3.3. GRAPH NEURAL NETWORKS

The first graph neural network model was introduced in [38] where it was used for feature propagation. This feature propagation was done by taking the features of a node and all its neighbors and feed this to a learnable function that projects the inputs onto a space with the same dimensionality of the input features. The weights of the learnable function are defined per node. Depending on the type of task the GNN is designed for, the output function of the GNN can either be defined on node-level or graph level. This design of the GNN is rather flexible and can be adapted for different tasks like node classification, graph classification, link prediction, and regression tasks.
Graph convolutional neural networks were first used in [9] where convolutional filters based on spectral graphs filters were introduced for graph signal processing. This work is based on an existing method that allows for a spectral operator on graphs using convolution [39].
Another method for GCNN was introduced in [11] where the authors design a convolutional filter based on the shift operator of the graph. This work presents two GNN architecture that implements pooling, unlike the work done in [9] which graph coarsening and clustering are used instead. The result is a more general type of architecture similar to the original CNN and thus is more easily adapted and expanded to other purposes.
The work in [45] presents graph attention networks (GAT) which use self-attention layers in their architecture. The purpose of those layers is to implicitly assign weights to different nodes in a neighborhood which makes the architecture more resistant to faults in the graph. This property is helpful for this thesis as we cannot guarantee the quality of our generated edges.

To better compare different architectures, the authors of [16] deemed it useful to introduce a framework to define different GNN architectures using a standard notation. This resulted in the EdgeNet, an architecture that allows individual nodes to explicitly assign weights to each of their different neighbors. Using this as a foundation, it is possible to rewrite all aforementioned architectures in the form of an EdgeNet. This paper also introduces the graph convolutional attention network (GCAT) architecture which combines the attention mechanism with graph convolution. GCAT has a large degree of freedom in learning parametrization compared to the GCNN while being more restrictive than pure EdgeNet.

3.4. GRAPH NEURAL NETWORKS IN CYBER SECURITY

In this Section we discuss other applications of graph neural networks in the domain of cyber security. Graph neural networks are useful tools for several graph-related tasks and there are cyber security-related problems that can be seen as graph-related tasks. For example, a network topology can be easily modeled as a graph, and finding an infected device within that network can be seen as a node classification task. On the other hand, there are also less obvious ways to use GNNs in cyber security. Especially of note

are models which need to generate a graph using an input that one would not initially consider as a graph. These models are of interest to us as we are trying to do something similar by generating graphs based on traces.

One possible application is the use of GNN's to detect vulnerabilities in code [53]. The authors propose a model which encodes the program control and data dependency of a program into a graph. The graphs are fed to a GNN which performs graph-level classification to determine which programs contain vulnerabilities. The GNN module is also able to extract useful features from the graph representations. This means that the GNN cannot only detect whether a program contains vulnerabilities but also gives information about the vulnerability. A similar idea is used to detect malware instead of vulnerabilities [49]. In this method, the binary of the (potential) piece of malware is represented as a control flow graph. These graphs are used as input in a graph convolutional neural network to perform graph-level classification. Memory forensics is another branch of cyber security in which Graph Neural Networks can be applied. The work in [40] shows a method that uses GNNs for kernel data structure detection. This is done by modeling the objects from raw memory dumps as a graph. In this graph each node represents a segment of contiguous memory bytes between two pointer fields. The edges represent either adjacency or a 'points-to' relation between two nodes. The goal of this GNN is first to generate higher-level features for the nodes after which the nodes are classified. Note that unlike most node-classification problems, which are often applied to one graph, this model aims for robustness and being able to classify nodes in multiple graphs. Notice that models proposed in [53],[49],[40] all represent a piece of software/code as a graph. A graph is a useful and versatile model as it can show the different relations of the operations in the software. For example, operations can be related by shared variables or by the control flow of the software.

Another application is anomaly detection in an IoT network [35]. The authors propose a distributed system that represents an IoT environment modeled as a graph. The distributed system runs a GNN in real-time to perform node/edge classification as an anomaly detector. The system is distributed to safeguard against compromised nodes and spread the expense of the calculations over the entire network. There have been already some studies on using GNNs for social network-related tasks [10],[4],[50]. The work in [6] shows that GNNs can also be used for cyber security purposes. The authors use the GNN for anomaly detection, or to be more precise, to find anomalous profiles focusing on spam accounts. Unlike the other works mentioned in this section, this work does extensive feature pre-processing instead of letting the GNN generate high-level features. The work gives some rationale for this, namely because these pre-processed features are connected to hypotheses about properties of anomalous profiles the authors wanted to test. Nevertheless, the full power of the GNN remains unutilized as the authors could have used the generated features to deduce beneficial properties and compare these against their handcrafted features.

For networks consisting of either users or devices, graphs are an obvious representation. This network will generally not be fixed but change dynamically over time when devices are added and removed. The propagation aspect of a graph is more important

for a computer network as malware tends to spread from a single node (device or user) to the entire network.

3.5. DISCUSSION

In this Chapter, we summarize the relevant literature related to the subjects of our thesis. In Section 3.1 we discuss non-profiled power analysis, which is the direct predecessor of profiled-side channel analysis. In Section 3.2 we examine works related to profiled side-channel analysis. We begin with works related to the template attack, which is the first iteration of the profiled SCA paradigm. Next, we discuss countermeasures that were implemented to make side-channel analysis (both profiled and non-profiled) more difficult. We continue by looking at works that use machine learning methods to solve side-channel analysis problems. We finish by discussing how deep learning is used to solve SCA, which is the current state-of-the-art method. In Section 3.3 we discuss works related to graph neural networks. First, we take a look at the work that introduced the concept of graph neural networks. Next, we examine several works which are about convolutional graph neural networks. In Section 3.4 we look at other applications of graph neural networks in the domain of cyber security. Those works model the execution flow of software and computer networks as graphs used in conjunction with GNNs.

The goal of this thesis is to use a graph neural network to solve a side-channel analysis problem. We aim to do this by transforming our SCA problem in a graph signal classification problem. This requires us to generate a graph on which the signal is propagated. We shall treat the power traces as graph signals for this purpose. From this, it follows that the vertices in the graph represent the time instances (i.e., features) of the traces. We generate the edges of the graph based on the correlation between the measurements for each pair of features. Given this translation from an SCA into a graph signal classification problem, we use a graph neural network to solve the problem and compare the results with state-of-the-art. We experiment with different architectures for the GNN (GCNN, GCAT, EdgeNet) to see how they differ in performance. Furthermore, we also want to observe how our model deals with the different leakage models (HW, IV) and if there is a large performance gap between them. We also want to see how graph neural networks deal with countermeasures such as masking and desynchronization. Finally, we perform hyper-parameter analysis to see whether we can reduce the number of learnable parameters without substantially reducing the performance of our model. We hope to find out if using graph neural networks is a viable alternative for the currently used models for side-channel analysis by asking those questions.

4

TRANSLATION FROM SIDE-CHANNEL ANALYSIS TO GRAPH SIGNAL CLASSIFICATION

In this Chapter, we describe our approach to translate the SCA problem into a GNN classification problem and answer Research Question 1. In Section 4.1 we explain which function we use to the traces to generate a graph. The nodes of this graph correspond with the features of the traces and the edges signify the relationship between the features. Subsequently, we run a graph neural network over the built graph. In Section 4.2 we show some examples of the results of our proposed method to generate graphs.

4.1. GRAPH SIGNAL CLASSIFICATION

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a feature vector $\mathbf{x}$. The number of features equals the number of vertices $|\mathbf{x}| = |\mathcal{V}|$ so each feature $x_v \in \mathbf{x}$ is assigned to corresponding vertex $v \in \mathcal{V}$ and $\mathbf{x}$ can be considered a graph signal defined on the vertices of graph $\mathcal{G}$. We can now consider a graph signal classification problem which requires a classifier that, given graph G and a graph signal $\mathbf{x}$ as input, outputs a label prediction y for graph signal $\mathbf{x}$. So each vertex in the graph represents a time instance. This implies each edge represents the relationship between two time instances. The feature vector (or graph signal) is a a trace $\mathbf{t}_i$. Since a trace is a time series, each feature f_j represents the measurements taken at time instance j. As we are talking about the relation between two sets of measurements, the (Pearson) correlation is a reasonable metric to determine the relationship (i.e., generate edges) between two time instances (i.e., nodes). We chose correlation since it actually calculates the similarity between two sets of repeated measurements. Most other candidate metrics, like Euclidean or Cosine distance, instead measure the distance between a set of points.

Given the Pearson correlation as a similarity metric, there are still multiply methods to build a graph based on the correlation between a pair of time instances. One method would be to create a fully connected graph and let the weights of each edge correspond to the correlation between the nodes. This method is likely to introduce complications from a large number of edges. The weights might interact unexpectedly, especially as the graph is expected to contain negative weights which correspond to a negative correlation.
Another method is to draw edges if the correlation between two nodes is above a certain threshold. This would lead to a smaller number of edges and ensures that each edge represents a meaningful relation. This method does have a few disadvantages. A slight variation in the threshold value could lead to both a significant change in the set of edges, or it might not change the set of edges at all. So it is a bit finicky. Furthermore, this also means that it is difficult to predict how many edges it will generate.
The final method we introduce is based on the ($k-$)nearest-neighbor principle. This is to say; each node n is connected with its k nearest neighbors. I.e., edges are drawn towards the k nodes which have the highest correlation with node n. This method has the advantage that the number of edges is stable and that each node is guaranteed to be connected to at least k other nodes. The disadvantage is that this method may leave out edges corresponding to high correlation connections or include edges between nodes with low correlation. In the end, we chose the latter approach as we believe that having a guaranteed number of edges is more important than each edge representing a strong relationship between the nodes. Especially since we are not confident that correlation is the most representative metric to describe the relation between the nodes.

The mapping function $\mathscr{F}$ in Algorithm 1 shows how to generate the graph for this problem from a set of training samples. The number of time instances Q is independent of the size of the training data but is determined by the output of the feature reduction. For a pair of time instances (i, j), an edge e_{ij} is generated when the correlation coefficient between their measurements is in the top n_e of all possible edges for that node.

Algorithm 1 GenerateGraph($\mathbf{T}, Q, n_e$)

$\mathscr{E} \leftarrow \emptyset$ ▷ Initiate edge set
$\mathscr{V} \leftarrow \{i, ..., Q\}$ ▷ Map every time instance to a vertex
$\text{cor}(\mathbf{v}, \mathbf{u}) = \frac{\sum_{i=1}^{n}(v_i-\bar{v})(u_i-\bar{u})}{\sqrt{\sum_{i=1}^{n}(v_i-\bar{v})^2}\sqrt{\sum_{i=1}^{n}(u_i-\bar{u})^2}}$ ▷ Define correlation function
▷ $\bar{x}$ is the sample mean of $\mathbf{x}$

for $i = 1 : Q$ **do**
  for $j = 1 : Q$ **do**
    $cv_{ij} \leftarrow \text{cor}(\mathbf{f}_i, \mathbf{f}_j)$ ▷ Calculate correlation for each pair of nodes
  end for
  Let $\mathbf{n}_i$ be the top n_e nodes j for which cv_{ij} is highest.
  $\mathscr{E} \leftarrow \mathscr{E} \cap (i, \mathbf{n}_i)$ ▷ We generate the edges for node i and add them to our set of edges
end for
$\mathscr{G} \leftarrow (\mathscr{V}, \mathscr{E})$
return $\mathscr{G}$

The resulting graph can be used to run on top of it a classification GNN as described in Section 2.4. The output $\mathbf{y}$ represents the classification distribution. We use cross-entropy as a loss function defined as

$$CE(\mathbf{Y}, \hat{\mathbf{Y}}) = -\frac{1}{P}\sum_{n=1}^{P}\sum_{c=1}^{C} y_{nc} log(\hat{y}_{nc}) \tag{4.1}$$

where $\hat{\mathbf{Y}} \in [0,1]^{C\times P}$ represents the predicted posterior distribution obtained from the GNN. P denotes the number of training traces and C denotes the number of classes. $\mathbf{Y} \in \{0,1\}^{C\times P}$ represents the ground truth for the classification. Algorithm 2 outlines the entire training process for the GCNN. The other architectures are trained using a similar process, but with a differently sized $\mathbf{\Theta}$ and different convolution operations. We simplify the notation a bit by considering the MLP as a function to transform the output signal into a classification vector. In reality, the MLP is also a neural network containing trainable parameters trained using stochastic gradient descend. However, we opt not to show this to focus on the operations which are more relevant to our research.

Algorithm 2 TrainGCNN($\mathbf{T}, \mathbf{k}, c, L, K, F$)

$(\mathbf{T}', Q) \leftarrow$ ReduceFeatures($\mathbf{T}$) ▷ Apply feature reduction to the dataset
$((\mathbf{T}_{train}, \mathbf{k}_{train}), (\mathbf{T}_{test}, \mathbf{k}_{test})) \leftarrow (\mathbf{T}', \mathbf{k})$ ▷ Split dataset in test and train
$\mathcal{G} \leftarrow$ GenerateGraph($\mathbf{T}_{train}, Q', c$) ▷ Generate the graph
$\mathbf{S} \leftarrow \mathbf{A}$ ▷ Let the shift operator be the adjacency matrix of G
$\mathbf{\Theta} \leftarrow$ rnd$^{F\times K\times L}$ ▷ Initialize the parameter matrix with random values
$\mathbf{X}_0 \leftarrow \mathbf{T}_{train}$
while Not Convergence **do**
  for $l = 1 : L$ **do**
    $\mathbf{X}_l = \sigma(\sum_{k=0}^{K} \mathbf{S}^k \mathbf{X}_{l-1} \boldsymbol{\theta}_{lk})$ ▷ Calculate the output for each layer
  end for
  $\mathbf{Y} =$ MLP($\mathbf{X}_l$))
  Calculate loss $\mathscr{L}$ using cross entropy
  Update $\mathbf{\Theta}$ by stochastic gradient descend
end while
Return $\mathbf{\Theta}$

Considering we are talking about time instances, another logical method would be to simply connect each time instance j with $j+1$ as they are temporally connected. This type of graph is known as a cyclic graph. However, this method is not able to translate more complex relationships. For example, if it is known that a certain interesting peak happens somewhere in the interval $[j-5, j+5]$, this would show up as a heavily connected neighborhood in using the first method. In contrast, the latter method would not encode this information. Furthermore, if one applies feature selection, the selected features are not equidistant in temporal space. Which is to say, if points $i-a, i, i+b$ before feature selection correspond with points $i-1, i, i+1$ after feature selection, a is not guaranteed to be equal to b. Thus, one might connect two nodes with an edge while those nodes are quite far apart in the original dataset and this edge would be equally impor-

tant as an edge between two nodes that are actually close.

4.2. EXAMPLES

This subsection shows some examples of how the graphs resulting from our graph generation method look like. We visualize the graphs by showing the resulting adjacency matrices. We've chosen not to show the actual graph visualization as a large number of nodes (especially for the ASCAD dataset) makes the graph look rather messy and complicated to see what exactly is going on.

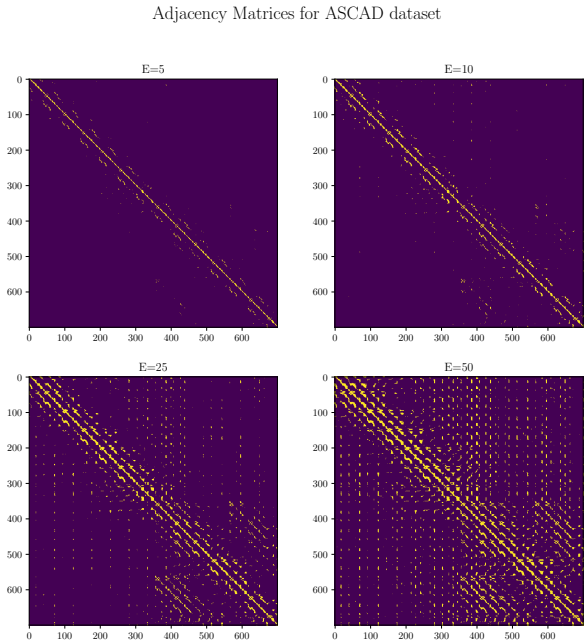

Figure 4.1: Visualization of the adjacency matrices generated by our method using the ASCAD dataset for different numbers of edges per node

In Figure 4.1 we show an example of a graph generated using the method explained in Algorithm 1 for the ASCAD dataset. The diagonal line in the resulting graph for $n_e = 5$ shows that most nodes are connected towards other nodes whose corresponding time instances are close. I.e., the resulting graph resembles a cyclic graph. This is not very surprising because the original data is a time series. So it is reasonable to expect that two subsequent time instances have similar values and are therefore correlated. When we increase the number of edges per node, we see that more edges start emerging at random places. Nevertheless, it is not entirely random since those edges are arranged in a

somewhat grid-like structure. These lines likely correspond with more essential features, although it could also be a coincidence. We also see that despite the emergence of those edges, the density of edges is still getting higher across the diagonal line corresponding with the cyclic graph. When considering which of these graphs to use for in the GNN, it seems that the ones with the higher number of edges have more edges that are randomly generated. Or rather, those edges likely correspond with little to no correlation between the node features. Therefore, the graph $n_e = 5$ seems to be the best candidate graph to use in our model since it contains a meaningful pattern without too many arbitrary edges.

In Figure 4.2 we show several examples of graphs generated by our method for the DPAv4 dataset using feature reduction. Compared to the ASCAD graph, these graphs have a far lower resemblance to the cyclic graph. The diagonal line is present since all nodes are also connected to themselves. This is likely since we've applied feature reduction to our dataset. So features which are close (in the temporal sense) after feature reduction are not necessarily close in the original data. This explains how the subsequent features are less likely to correlate when using feature reduction. Despite these graphs looking less like a cyclic graph, we can still distinguish some patterns and clusters. For example, we see that clusters emerge around the $(3,3)$ area and the $(10,10)$ area. This indicates that at the beginning, the traces are more similar and start decorrelating more over time. While none of the graphs look like they have a meaningful pattern, one could say that the graph for $n_e = 2$ looks rather sparse. On the other hand, the adjacency matrices for $n_e = 8$ and $n_e = 10$ seem rather overcrowded. So the graph for $n_e = 5$ looks like an adequate candidate to use in that regard.

4.3. CONCLUSION

In Section 4.1 we describe the method we use to generate graphs and translate the SCA problem to a GSP problem. In Section 4.2 we show visualizations of the generated graphs. This Chapter answers Research Question 1 as we show how we transform a SCA problem into a GSP problem. We consider each time instance in the traces as a node for our generated graph. We generate the edges of the graph based on the correlation between features corresponding to the time instances.

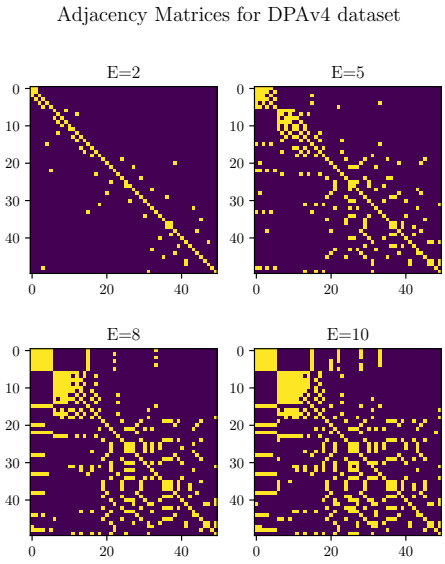

Figure 4.2: Visualization of the adjacency matrices generated by our method using the DPAv4 dataset for different numbers of edges per node

5

NUMERICAL EXPERIMENTS

In this Chapter, we present our experiments and their results. In Section 5.1 we discuss the two datasets we are using for our experiments. In Section 5.2 we consider applying feature reduction to the traces. In Section 5.3 we show the configuration of our experimental setup. In Section 5.4 we discuss the main findings of our experiments. Which is to say, how well did our model perform on both datasets. In Section 5.5 we examine how well our model deals with countermeasures and analyze the behavior of our model more closely. In Section 5.6 we look at the ablation studies we performed. In Section 5.7 we investigate the use of different values for chosen hyperparameters. In Section 5.8 we discuss all the results that we have gathered and conclude this Chapter.

5.1. DATASETS

In this Section, we discuss the datasets we use in our experiments. Both datasets contain measurements during the Sbox operations of the AES cipher. Both measured devices used a fixed key for the data generation: the used key is the same for each trace.

5.1.1. DPACONTEST V4

We use the DPAcontest v4 (DPAv4) [37] dataset for the exploratory phase of our experiments as this is the most straightforward dataset commonly used for side-channel analysis. This should be understood in this case as being unprotected, containing little noise and only the samples of the relevant operation are selected. Unprotected means that no countermeasures to prevent SCA are present which we limit to masking and desynchronization in this thesis. The traces are obtained by measuring the power usage of an AES-256 implementation on an Atmel ATMega-163 smart card.

While the AES implementation used to generate the traces is masked, the masks are given and therefore the dataset can be treated as unmasked. The dataset contains 50,000 traces that consist of 10,000 features and includes measurements of a masked AES implementation, but it can be converted to an unprotected implementation by assuming the mask to be known. However, the mask used in DPAv4 is known to be flawed and it

leaks[28]. Accordingly, the leakage model changes to

$$y(t, k^*) = HW(\text{Sbox}[p \oplus k^*] \oplus M), \tag{5.1}$$

where M represents the known mask. We apply feature reduction to this dataset by selecting 50 features with the highest absolute correlation coefficient between the measurement traces and the corresponding keys. Since we know that this dataset is easily breakable, we apply the feature reduction to this dataset to increase the speed. This allows us to perform tasks that require a large number of runs like hyperparameter tuning.

5.1.2. ASCAD

The ASCAD dataset [1] contains electromagnetic radiation measurements of a masked AES-128 implementation on an 8-bit AVR microcontroller (ATmega8515). The ASCAD dataset contains 60,000 traces of 700 features, split into a profiling set of 50,000 traces and an attack set of 10,000 traces. So it is faster to run tests on the dataset without feature reduction compared to the DPAv4 dataset. It contains countermeasures to test against to see if our model also holds for a more realistic scenario. As we want to treat ASCAD as a masked model, we use the leakage model in Equation (5.1). The ASCAD dataset also contains options to differ the degree of desynchronization used in the traces. The degree of desynchronization d_{max} signifies that each trace (both the training set and attack set) has been shifted by $\delta \in [0, d_{max}]$ to the left.
Initial tests have shown that using cross-evaluation for experiments that use the ASCAD dataset is not feasible due to time and resource constraints. Furthermore, the guessing entropy did not converge to 0 after 1000 traces. Instead, we randomly sample 10,000 traces from the set of profiling traces for each instance, from which we use 9000 traces for the training set and 1000 traces for the validation set. For the testing set, we use all 10,000 traces from the set of attack traces,

5.2. FEATURE REDUCTION

Each trace contains many samples, so feature reduction might be desirable to break the "curse of dimensionality". If we apply feature reduction to the traces, this would help to reduce the complexity and runtime of our problem. Whether we want to use feature reduction depends on the dataset we are using. Let us consider the datasets we want to use: the DPAv4 dataset and the ASCAD dataset. The DPAv4 dataset contains 10,000 samples per trace and is noted to be relatively easy to break. Both factors make it appealing to use feature reduction on the DPAv4 dataset, as there is much room for speedup regarding the number of features. Since the dataset is easy, the potential performance loss is not as much of an issue. In fact, with such a large number of features, feature reduction may even increase the performance.
For the ASCAD dataset we use only 700 samples[1] per trace and is more difficult to break than the DPAv4 dataset. Therefore we do not apply feature reduction to this dataset as 700 features allow for a reasonable runtime. If the results show that we can take a minor hit in performance, we may use feature reduction for the ASCAD dataset for later

[1] To be more precise, it has 700 features for processing the targeted key byte

experiments. In the related literature[5], [18] there are already established methods of feature reduction for the datasets we plan to use in the related literature: selection based on highest absolute correlation, and Principal Component Analysis (PCA).

5.2.1. ABSOLUTE CORRELATION

Absolute correlation is used as a feature selection method in [18]. This method selects F' features with the highest absolute correlation coefficient between the measurement traces $\mathbf{f}_j$ and the corresponding keys $\mathbf{y}_k$. The notation $\max^{F'}$ means that it selects the F' highest values, instead of a single value that max would denote.

$$\mathbf{X}' = \max_{j \in \{1,\dots,Q\}}^{F'} \sum_{k=0}^{K} \text{corr}(\mathbf{f}_j, \mathbf{y}_k) \tag{5.2}$$

Matrix $\mathbf{X}' \in \mathbb{R}^{P \times F'}$ represents the reduced data matrix after the feature reduction, where F' is the number of reduced features. $\text{corr}(\cdot,\cdot)$ represents the Pearson Correlation between two vectors of equal order. $\mathbf{y}_k \in \{0,1\}^P$ represents the classification vector of class k, where entry y_{tk} signifies whether trace t is associated with key k.
The main advantage of this method is that it leverages the relation between the information in the data and the target keys so it can filter out noisy and irrelevant features. A downside of this method is that it does not care about the temporal distance between the features. When considering that there are different regions that contain relevant features, this method might ignore some of these regions while sampling a large number of features from other regions. Furthermore, this method does not guarantee that the resulting features are equally good discriminators for all classes. This method also assumes a direct correlation between the data and the classes, which is not the case for the ASCAD dataset [1].

5.2.2. PRINCIPAL COMPONENT ANALYSIS

Principal component analysis (PCA) is used in [1] as feature reduction method. PCA has some disadvantages which makes it a poor choice as a feature reduction method for this approach. PCA transforms the data instead of just selecting the features of interest. This means the data cannot be interpreted as a time series anymore and therefore loses the temporal information embedded in the data. Thus, PCA is only helpful for feature reduction if we do not care about the temporal relationship present in the original data. In general, there are other structures in the traces which might disappear when using PCA. Therefore, we shall not use PCA as a feature reduction method but instead use the absolute correlation method.

5.3. EXPERIMENTAL SETUP

We use two different datasets in our experimental setup: the DPAcontest (DPAv4) dataset[37] and the ASCAD dataset[1]. From those dataset we select $N = 10{,}000$ traces to use for cross-validation with $k = 10$. In each experimental run, we split the selected dataset into a train set of $N = 9000$ and test set $N = 1000$ using ten-fold cross-validation. As shift operator we use the adjacency matrix $\mathbf{S} = \mathbf{A}$. The graph over which the GNN-architecture is run will be generated using Algorithm 1 using $n_e = 5$ as the number of neighboring edges

generated for each node. We train the neural network as described in Algorithm 2.
We perform experiments using the GCNN, GCAT, and EdgeNet architectures as described in Section 2.4. We chose the GCNN as our primary GNN architecture as it is the faster architecture, which makes it the most suited for repeated experiments. We use the other architectures introduced in Chapter 2 in experiments which are designed to compare their performance. We also introduce an architecture that uses an edge varying filter in the first layer and a graph convolutional filter in the second layer. The main idea behind this method is as follows: The first layer functions on a more local level as it can discriminate between edges. The second layer functions on a global level and introduces stability as it is less likely to overfit. So in theory, this architecture should combine the best properties of both the GCNN and EdgeNet. We shall denote this architecture as GC-EdgeNet in the remainder of this thesis.
For each different architecture, we perform hyper-parameter search to find the best combination of hyper-parameters for each architecture. We do this by performing a grid search over the following hyper-parameters: number of features $F \in \{1,2,4,8,16,32,64,128\}$; filter order $K \in \{1,2,3,4,5\}$; and number of layers $L \in \{1,2,3,4\}$
For the leakage model, we use both the Hamming Weight and the Intermediate Value leakage models. We use the Hamming Weight(HW) of the intermediate value for the leakage model as described in Equation (5.1) for our initial experiments. This means that we consider a classification problem with 9 classes when using the HW leakage model. We directly use the intermediate value for the leakage model for other experiments, as described in Equation (2.1). This implies a classification problem with 256 classes when using the IV leakage model.
The classification MLP has two fully connected layers: the input layer has $F \times Q$ nodes and the output layer has C nodes. Where F is the number of features used by the output signal $\mathbf{X}_l$, Q is the number of nodes in the graph, and C is the number of classes. So the number of learnable parameters for this MLP is $F \times Q \times C$.

Furthermore, we train the GNN using ADAM[19] as optimiser with learning rate 0.001 and forgetting factor $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Training is performed in 100 epochs with a batch size of 100. As an evaluation metric, we use the guessing entropy as described in Section 2.3.2.
The main goal of each experiment is to observe how many traces are required to converge the guessing entropy to zero. We visualize this by plotting the guessing entropy against the number of traces for each experiment. We also give the numerical results in a table for a more precise comparison. This also allows us to show much variance there is between different runs in an experiment. This table contains the mean number of traces required for the guessing entropy to get below a threshold. It also includes the standard deviation and highest and lowest number of traces required to reach the threshold for the instances of the given experiment.
We compare our results with the work of [51] and [1]. The work in [1] serves as a baseline while the work in [51] is more recent. We give the number of traces to reach a guessing entropy of 1 and the number of trainable parameters in Table 5.1 for easy reference.

Dataset	$\|\Theta\|$	N_t
DPAv4	52112	4
ASCAD	66,652,444	1146
ASCAD(desync)	66,652,444	>5000

(a) [1]

Dataset	$\|\Theta\|$	N_t
DPAv4	8782	3
ASCAD	16960	191
ASCAD(desync)	87280	244

(b) [51]

Table 5.1: Table which shows the number of learnable parameters $|\Theta|$ and number of traces to reach a guessing entropy of 1 N_t for the models used in [1] and [51]

5.4. MAIN FINDINGS

In this Section, we want to establish the performance of our model. We want to see how well it performs on both the DPAv4 and the ASCAD datasets. We also want to compare the different architectures we've discussed in Chapter 2 on the different datasets. While we are primarily interested in performance (i.e., how many traces are needed to make the Guessing Entropy converge towards 1), we want to balance this against the complexity of the model.

The complexity of a model is directly related to the number of learnable parameters in a model. In Table 2.1, we can see that each architecture has a different dependency between hyperparameters and the number of learnable parameters. So reducing the number of learnable parameters has two sides: both the choice of architecture and the values of the hyperparameters are important. Furthermore, the choice of the leakage model is also relevant for the number of learnable parameters since the size of the classification MLP is proportional to the number of classes. As mentioned earlier, the number of classes when using HW as leakage model is 9, while the number of classes when using IV as leakage model is 256. The number of learnable parameters for the classification MLP is linearly proportional to the number of classes. When the other values are equal, the IV model has more than an order of magnitude more learnable hyperparameters than the HW model. This means that we cannot answer research question RQ2(a) without considering the answers to RQ2(c) and RQ2(d) as well.

5.4.1. DPAv4

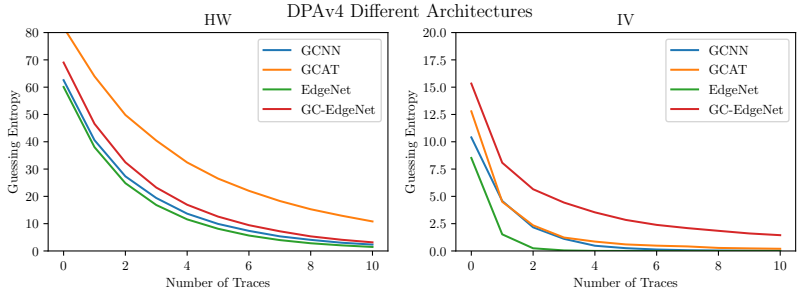

Figure 5.1: Results using different architectures using the DPAv4 dataset for different leakage models. (left) Hamming Weight leakage model; (right) Intermediate Value leakage model

Architecture	μ	σ	min	max
GCNN	13.4	2.3	10.0	24.0
GCAT	24.64	3.8	18.0	33.0
EdgeNet	11.72	1.7	9.0	18.0
GC-EdgeNet	14.55	2.9	11.0	27.0

(a) HW

Architecture	μ	σ	min	max
GCNN	3.60	0.76	2.0	6.0
GCAT	4.25	4.44	2.0	18.0
EdgeNet	1.62	0.49	1.0	2.0
GC-EdgeNet	15.83	14.79	3.0	97.0

(b) IV

Table 5.2: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

The results in Figure 5.1 and Table 5.2 show that the GCAT is the worst performing architecture while the EdgeNet is the best performing architecture for both leakage models. The GC-EdgeNet architecture performed even worse on the IV leakage model, but this is likely due to an outlier. The main difference between the GCAT and the other architectures is that it incentives to reduce the weight of all but the most important edge in the neighborhood for each node. If this is a problem where strong inter-connectivity between nodes is essential to the solution, this would be a disadvantageous property of the GCAT. As our graph is generated based on heuristics rather than ground truth, it is unlikely that there is an edge that is considerably more important than the other edges in a neighborhood.

The results show that the IV leakage model has a considerably better performance than the HW leakage model. We have three possible explanations why this is the case: The first explanation would be that the IV model does not suffer from class imbalance, unlike the HW model as described in [18]. Another explanation would be that the IV model better describes the leakage compared to the HW model. The final explanation would be that the architecture that uses the IV model is more powerful because the classification MLP is more complex. This complexity is derived from having more learnable parameters since the IV model uses more classes. This would imply that the MLP is the main contributor towards the performance rather than the graph filter layers.

The first explanation seems rather unlikely since the difference in performance between balanced and unbalanced datasets (for comparable configurations) [18] is a lot less than the difference we observe in our experiments. The second explanation is also not very likely as this would contradict earlier literature [27],[36],[30],[15],[25]. Therefore, the third explanation is the most likely. We can test the last explanation by observing how much the MLP contributes to the model in the ablation studies.

When we compare our results for the IV leakage model with the results of the works in Table 5.1, we observe that the EdgeNet architecture requires fewer traces to converge than both other works. However, the number of parameters of the EdgeNet architecture is higher than for the work of [51]. As the classification MLP contains a considerable share of the learnable parameters for the IV leakage model, it is impossible to significantly reduce the number of learnable parameters[2]. Meanwhile, the performance of the

[2] Unless we change the design of our model to replace the classification MLP with something else. But that would be outside the scope of this thesis.

HW leakage model is considerably worse even though it has a more reasonable number of learnable parameters.

5.4.2. ASCAD

Figure 5.2 shows that the performance for the ASCAD dataset is rather disappointing. In fact, from all the architectures only the GCNN can produce results for which some instances of our experiment converge. Therefore, we will also show the results where the non-convergent instances are filtered out. We also give the percentage of convergent instances for each setup.

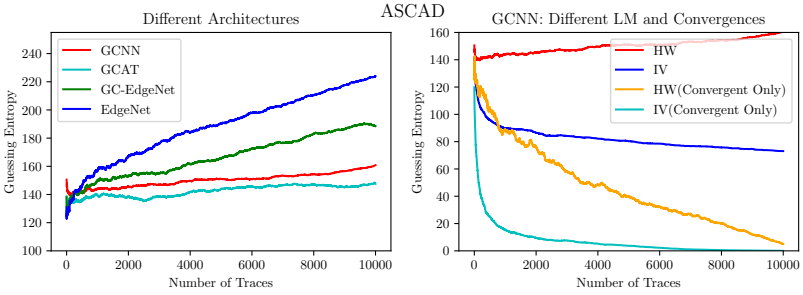

Figure 5.2: Results using the ASCAD dataset for different leakage models. (left) Different architectures using the HW leakage model; (right) Different Leakage models and ignoring non-convergent results using the GCNN architecture

Leakage Model	μ	σ	min	max	%conv
HW	9935.7	195.9	9348.0	10001.0	10%
(Conv. Only)	9348.0	0.0	9348.0	9348.0	-
IV	6181.91	4460.40	242.0	10001.0	45.45%
(Conv. Only)	1599.0	2294.28	242.0	6157.0	-

Table 5.3: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound and the percentage of convergent runs. Runs which did not converge below this threshold are given the value 10001. All results are only for the GCNN architecture, as none of the other architectures has a convergent instance.

Based on the results of the DPAv4 dataset, it was not unexpected that the GCAT architecture performed poorly. It is a bit surprising to see that the EdgeNet architecture had a bad performance as well as it was the best performing architecture for the DPAv4 dataset. The most likely explanation is that the EdgeNet architecture is too complex when using the ASCAD dataset. To be more precise, the number of learnable parameters for the EdgeNet architecture is dependent on the number of nodes and edges in the graph. This number is a lot higher for the ASCAD dataset compared to the DPAv4 dataset, for which we performed feature reduction, as we can see in Table B.2.
We also see that the IV leakage model has a far better performance than the HW leakage model for the ASCAD dataset. When we look at the number of learnable parameters in

Table B.2, we can calculate that for the GCNN, the number of learnable parameters of the MLP is larger than the number of learnable parameters for the graph filter layers. This is a strong hint for our explanation that the MLP is the main contributor to the performance in our model.
When we compare our results with the results in Table 5.1, it is evident that our results are a lot worse compared to both works. Even if we disregard that a significant fraction of our instances does not converge to zero, the number of traces required for convergence is still higher for our model than the other models.
When we use the results in this section to answer our research questions, let us consider both datasets. When we look at the ASCAD dataset, the answer to RQ2(d) is rather straightforward: the GCNN architecture is the only one that produces convergent results, so it is the only suitable architecture for that dataset. When we look at the DPAv4 dataset, the answer to this question gets a bit more complicated. Concerning performance, EdgeNet and GCNN have similar performance, although EdgeNet has a slight edge. However, the EdgeNet has fewer learnable parameters while the GCNN has a faster runtime. Furthermore, the GCAT architecture has the fewest parameters but also has worse performance. When we consider RQ2(c), we see that the IV leakage model performs better than the HW leakage model for both datasets. However, this difference in performance is likely since the complexity of the classification MLP gets increased when using the IV leakage model. It should also be noted that our hyper-parameter search method only looked at the best performance. So it could be possible to have a configuration with a negligible difference in performance while having a far lower number of learnable parameters. We look more closely into this in Section 5.7 which gives us the answer for RQ2(a).

5.5. SECONDARY FINDINGS

In this Section, we want to take a closer look to see how good our model deals with countermeasures and to get answers for research question RQ2(b). First, we take a look at the masking countermeasure. We compare the results of the masked DPAv4 dataset to the results of the unmasked DPAv4 dataset. We look at the desynchronization countermeasure by comparing the regular ASCAD dataset with the desynchronized ASCAD dataset. So we try to answer RQ2(b) by splitting it into two subquestions: 'What is the difference in performance between a masked and an unmasked implementation?' and 'What is the difference in performance for a synchronized and a desynchronized dataset?'
Next, we investigate the causes for the bad performance of the ASCAD dataset. There are several possible causes for the bad performance, but there are two in particular that we want to investigate: one possible cause is that our model has difficulty learning a good solution for this problem. Therefor we want to analyze how our model learns utilizing the loss curves. We plot the cross-evaluation loss for the loss curve as in Equation (4.1) for both the validation and the training sets at each epoch. The desirable behavior would be for both the training loss and validation loss to converge towards zero. It is possible that our model could be overfitting on the training data. In that case, the training loss would converge to zero while the validation loss would not converge to zero.

Another thing that we want to investigate is why we do not have consistent conver-

gence behavior. A possible cause could be the random selection of the training set. It could be the case that some randomly generated training sets can train a better model compared to other sets. We test this by running an experiment where we used a fixed training set for each instance instead of a randomized one. If the choice of the training set influences the performance of our model concerning convergence, we expect to have a percentage of convergent instances of either 0% or 100%.

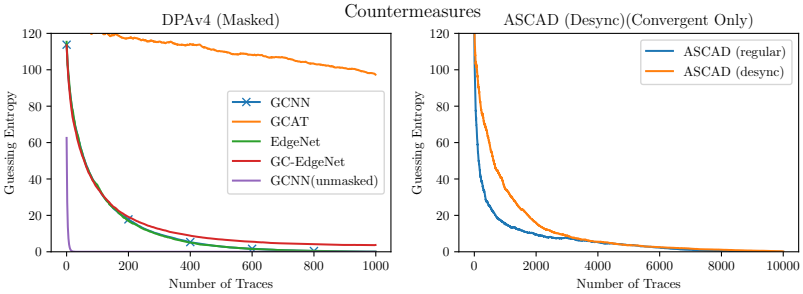

Figure 5.3: Results comparing countermeasures versus baseline performance. (left) DPAv4 dataset using the masking countermeasure against unmasked implementation; (right) ASCAD dataset comparing desynchronized dataset against synchronized dataset using only convergent traces.

Architecture	μ	σ	min	max
GCNN	498.7	222.8	174.0	929.0
GCAT	1001.0	0.0	1001.0	1001.0
EdgeNet	498.1	188.5	252.0	898.0
GC-EdgeNet	485.4	269.9	174.0	1001.0

Table 5.4: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

Dataset	μ	σ	min	max	% conv
Regular	1599.0	2294.28	242.0	6157.0	45.45%
Desync	2785.67	1924.68	1381.0	7019.0	54.5%

Table 5.5: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

We can see in Figure 5.3 and Table 5.5 that the masked implementation of DPAv4 is far more challenging to solve than the unmasked variant. The number of traces required for the guessing entropy to converge is about ten times higher than the number for the unmasked variant. It is interesting to note that the GCAT architecture cannot find a solution at all for the masked implementation (or rather, the curve suggests that it is converging very slowly). This is another argument that the GCAT architecture does not appear to be suitable for our problem. We observe that the EdgeNet and the GCNN appear to have similar performance. Since masking is a powerful technique and using GNNs does not seem to have inherent advantages to solving masked implementations,

it is logical that there is a significant difference between masked and unmasked.
The difference for the desynchronized dataset is a lot less pronounced. This can be explained by our use of graph filters and the structure of the graph. The desynchronized dataset diffuses the values for measurement over a larger range in time. Recall from Chapter 4 that the temporal structure of the traces remains intact for the graph that we generate. This means that our graph filters convolve over the neighbors which contain some of the diffused measurements. Therefore, when we convolve over our graph, the diffusion is counteracted to a certain degree. So our GCNN model contains an inherent property that makes it well suited to deal with the desynchronization countermeasure.

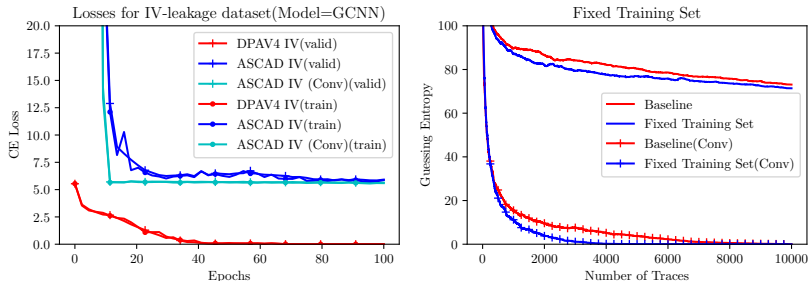

Figure 5.4: (left) Loss curves for different setups using the IV leakage model; (right) Comparison of using a fixed training set against using a random training set for each instance in an experiment. Both setups are run on the GCNN architecture using the IV leakage model on the ASCAD dataset.

Training Set	μ	σ	min	max	% conv
Random TrainSet	1599.0	2294.28	242.0	6157.0	45.5%
Fixed TrainSet	1067.0	794.6	284.0	2125.0	40%

Table 5.6: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are not included.

In Figure 5.4, we can see that the training and validation losses are very similar for each setup, so it is unlikely that we are overfitting. For the DPAv4 dataset, the losses converge to zero. As we mentioned earlier, this is the desired behaviour of the loss curve as it implies that the model has learned the dataset so that it can classify all the data correctly. Since the validation loss also converged, our model has not overfitted on the training data. More importantly, this also means that there is no inherent flaw or bottleneck in our model, making it impossible for the losses to converge. The loss curves for the ASCAD dataset do not seem to converge to zero. This is interesting, as one would expect that at the very least the training loss would converge to zero. So our model may be inclined to overgeneralize, making it difficult for the training loss to reach zero. Another explanation would be that the data is too complex/our model is not expressive enough for the training loss to converge. We see that our model is learning for about 15 epochs, after which the losses remain stable. The losses which correspond with only convergent instances show different behavior compared to the complete set. A possible explanation

for this would be that some instances are inclined to get stuck in local optima. To verify whether our model gets stuck in local optima, we will look into varying the learning rate of our model in Section 5.7. By varying the learning rate, our model traverses differently in the solution space, which might allow us to dodge those local optima.
In Table 5.11, we see a difference between the performance of the fixed training set and the random training set. However, the boxplot in Figure B.7 explains this difference as being caused mostly by an outlier in the results. More importantly, we see that the converge percentage is similar for both configurations. As we stated earlier, if being convergent was dependent on the selection of the training set, we would expect that the rate of convergence would be either zero or hundred percent when each instance uses the same training set. Since this is not the case, we can conclude that the differences in training sets do not explain the inconsistent convergences.

5.6. ABLATION STUDY

It is impossible to give a direct quality assessment of the generated graph as there are no metrics to measure the 'quality' of the graph. Furthermore, the quality of the graph is interwoven with the results of the GNN. Therefore, we want to perform an ablation study in which we examine the quality of the graph by comparing it with graphs which structure is not directly derived from the traces. We shall use four kinds of those graphs to compare:
The first one is the cyclic graph. A cyclic graph is defined such that for each node i there is an edge $(i, i+1)$ and for the last node N there is an edge $(N, 1)$. Note that the cyclic graph reflects the temporal relations which are present in traces. The next one is a random graph. For this graph, we randomly generate M number of edges, where M is the number of edges in the original generated graph. We generate a random edge by uniformly sampling two nodes $u, v \in \mathcal{V}$ and adding the edge (i, j) to our graph by updating the adjacency matrix $A_{ij} = 1$. The third one is a fully connected graph, where all nodes are connected to each other. The last graph we use is the unconnected graph, which uses the identity matrix as an adjacency matrix. We show a visual representation of each graph in Figure 5.5.

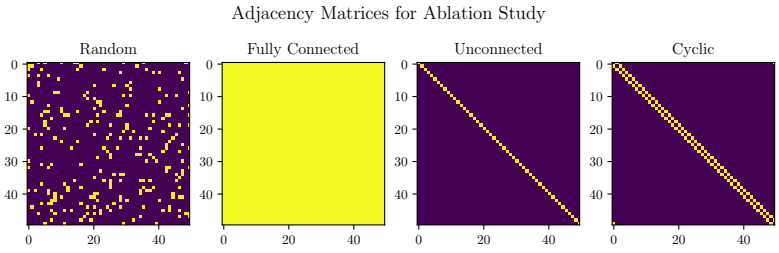

Figure 5.5: Visualization of the graphs used for the ablation study.

The cyclic graph aims to study the effect of the number of edges while retaining a meaningful relation (i.e., the temporal relation) between the edges. The purpose of the

random graph is to examine whether our graph is meaningful by removing the relational aspect of the edges while keeping the same number of edges. The fully connected graph represents maximum uncertainty about the relationship between the nodes. Finally, the unconnected graph aims to observe how much the classification MLP contributes to the architecture. The main idea behind this is that the graph filters layers do not learn anything from an unconnected graph, so the MLP is the only part of the GNN which actually learns.

When using the GCAT of EdgeNet, the experiment with the fully connected graph should be interpreted differently compared to the GCNN. GCAT/EdgeNet learn shift operator matrix $\mathbf{S}$ based on the input adjacency matrix $\mathbf{A}$. For a fully connected graph, each entry is $A(i,j) = 1$ for each vertex $i, j = 1, \ldots, N$. If we assume that $\mathbf{S}$ can be learned such that $S(i,j) \approx 0$ for any pair of vertices i, j, this implies that GCAT and EdgeNet are able to learn graph representations of the data. For this reason, we also include an additional experiment for those architectures where we add L1 regularization to the loss function of our model. The L1 regularization penalizes having large values for the learnable parameters. This encourages our model to learn more sparse graphs. The loss function using L1 regularization can be defined as:

$$CE(\mathbf{Y}, \hat{\mathbf{Y}}) = -\frac{1}{P} \sum_{n=1}^{P} \sum_{c=1}^{C} y_{nc} \log(\hat{y}_{nc}) + \lambda |\boldsymbol{\Theta}| \tag{5.3}$$

where $|\boldsymbol{\Theta}|$ is the sum of all learnable parameters and λ is the regularization parameter.

For the DPAv4 dataset, we perform the ablation test on each different architecture: GCNN, GCAT and EdgeNet. For the ASCAD dataset we only perform the ablation test on the GCNN architecture using the IV leakage model.

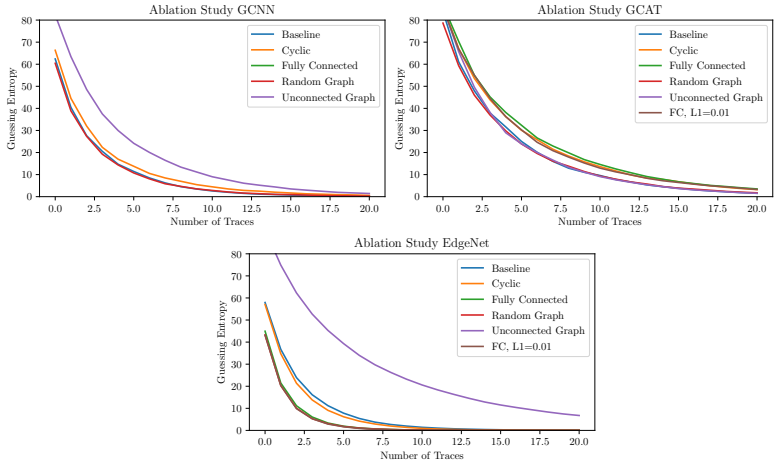

Figure 5.6: Ablation Study on the DPAv4 Dataset for different architectures

Graph type	μ	σ	min	max
Baseline	14.0	2.1	12.0	18.0
Cyclic Graph	16.8	4.2	11.0	25.0
Fully Connected Graph	921.5	132.07	634.0	1001.0
Random Graph	14.3	2.8	11.0	21.0
Unconnected Graph	22.8	3.8	18.0	30.0

Table 5.7: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound for the ablation study using the GCNN architecture. Runs which did not converge below this threshold are given the value 1001.

From the results in Table 5.7, we see that our model performs considerably worse on the fully connected graph. This is to be expected as a GCNN using an FC graph would convolve every node equally, resulting in oversmoothing. Secondly, the performance of the other graphs (baseline, cyclic, random) is a bit better compared to the unconnected graph. This shows that the classification MLP can solve the problem without using graph filter layers. Nevertheless, the differences in performance between the unconnected graph and the others show that the graph filter layers contribute something to the model. However, this difference is relatively small, so we must conclude that the classification MLP is a critical component of our model concerning performance when considering the DPAv4 dataset and the GCNN architecture.
It is noteworthy that the random graph has a similar performance as our baseline graph. This might imply that the quality of our graph generation method is not very good since it has comparable performance to a randomly generated graph. However, the contribution of the MLP seems to be dominant in this ablation study. Furthermore, we already noted that this dataset is easy to solve. So it is likely that the dataset is too easy for differences in the graphs to matter. In that case, any graph which is not sabotaging our model (like the fully connected graph) would be sufficient to help the MLP solve the problem. Since the ASCAD dataset is more difficult to solve, we expect that the differences in the suitability of the graphs will be more pronounced.
In Figure 5.6 we see the ablation study on the GCAT and EdgeNet architectures. This reinforces our belief that GCAT is not a good fit for our problem, as the unconnected graph has the same performance as our baseline. This means that the GCAT filters do not add anything useful to our model. The results for the EdgeNet ablation study are more fruitful. We see that both experiments using the fully connected graph have the best performance. This suggests that the EdgeNet can learn a graph that can outperform a pre-generated graph. An obvious disadvantage is that the number of learnable parameters (and run-time complexity) of the EdgeNet is dependent on the number of edges, which is the square of the number of nodes when considering a fully connected graph.

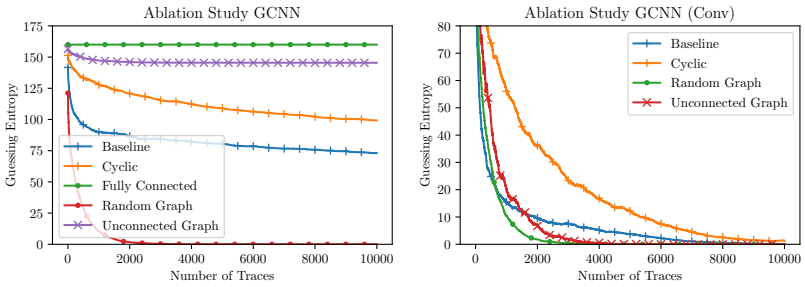

Figure 5.7: Ablation Study on the ASCAD Dataset for the GCNN architecture. (left) Ablation study using all instances; (right) Ablation study using only convergent instances

Graph type	μ	σ	min	max	% conv
Baseline	6181.91	4460.40	242.0	10001.0	45.5%
(Conv. only)	1599.0	2294.28	242.0	6157.0	-
Cyclic Graph	8711.67	2407.38	2369.0	10001.0	25%
(Conv. Only)	4843.67	1798.15	2369.0	6588.0	-
Fully Connected Graph	10001.0	0.0	10001.0	10001.0	0%
Random Graph	993.4	301.6	594.0	1646.0	100%
(Conv. Only)	993.4	301.56	594.0	1646.0	-
Unconnected Graph	9247.09	2384.07	1708.0	10001.0	9.1%
(Conv. Only)	1708.0	0.0	1708.0	1708.0	-

Table 5.8: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

Figure 5.7 shows the results of the ablation test for the ASCAD dataset using the IV leakage model. The most outstanding result is that the random graph manages to converge for all instances. Another important result is that the unconnected graph converges for some instances. This implies that the classification MLP is an important component in our model concerning performance. Compared to the DPAv4 dataset, the classification MLP appears to be less critical as the choice of the graph influences the performance and the number of convergent instances.
When we look at Table 5.8, we notice that the performance on the cyclic graph is worse, but otherwise, the baseline, random and unconnected graph seem to have a comparable performance. The performance of the unconnected graph being similar to the other graphs is a strong indication that the power of our model is derived mainly from the classification MLP. However, we can see that the unconnected graph does not converge as often as compared to the other graphs. So this means that the graph filters are somehow contributing to the performance.
It is of note that the random graph seems to be remarkably stable as it converges in every case. From a graph learning perspective, it is rather strange for a random graph to have better performance (with respect to convergent instances) than a graph with a structure that is more representative of the original data. While it could be argued that the graph

generation method we chose is not representative of the underlying structure, Figure 4.1 clearly shows a distinct pattern. Considering that the data we are using has a temporal structure and the pattern resembles a cyclic graph, the graph that we generate is at the very least a reasonable approximation of the underlying structure. This is to say; a cyclic graph is the graph representation of a time series. Nevertheless, while the random graph is devoid of inherent meaning, it still contains useful properties. The most noteworthy one is that the random graph is connected on a global scale as all edges are generated randomly. The graph that we generate is connected mostly locally as Figure 4.1 shows. If the data functions on a global scale, rather than on a local scale, one would expect a globally connected graph to perform better. So the performance of the random graph might indicate that data functions more on a global rather than local scale.
Another thing to consider is that the classification MLP seems to have a considerable contribution towards the power of our model. So it might be prudent to consider the MLP as the main component of our architecture and the GNN layers as a feature extraction component. In that case, we could consider the random graph filter as having a regularizing influence, as it combines random features and appears to improve the stability. The baseline graph filter has less of a regularizing effect. This is logical as we see that the baseline graph is connected chiefly locally. However, when we filter our non-converging instances, the baseline performs better, so it has more of an enhancing function.

5.7. HYPER-PARAMETER ANALYSIS

In this Section, we perform hyper-parameter analysis for several hyper-parameters. First, we perform hyper-parameter analysis on the DPAv4 dataset for the number of features F and the number of edges per nodes N_e . Next, we will perform hyper-parameter analysis for N_e for the ASCAD dataset and the loss rate LR. Finally, we try to increase the complexity of the ASCAD model by using higher values for the number of features F and layers L.

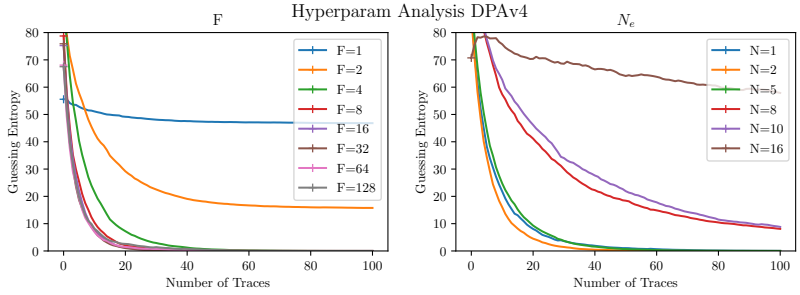

Figure 5.8: Hyper-parameter analysis on the DPAv4 dataset for number of features using the GCNN architecture F(left) and number of edges per nodes N_e (right)

F	μ	σ	min	max
1	909.1	275.7	82.0	1001.0
2	346.2	428.8	50.0	1001.0
4	40.8	8.8	28.0	55.0
8	25.3	7.2	16.0	43.0
16	21.2	4.4	15.0	28.0
32	19.8	3.6	16.0	28.0
64	23.0	8.0	15.0	42.0
128	27.3	13.6	15.0	56.0

N_e	μ	σ	min	max
1	40.1	18.4	19.0	85.0
2	31.2	9.1	21.0	53.0
5	43.3	10.5	23.0	64.0
8	219.0	262.8	94.0	1001.0
10	214.7	193.6	109.0	787.0
16	860.0	281.4	172.0	1001.0

Table 5.9: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

The results in Figure 5.8 show clearly that for very low values of F, the guessing entropy does not consistently converge to zero. It also shows that for the other values, increasing the number of features gives diminishing returns quickly . In fact, Table 5.9 shows that the performance of very high values of F decreases. This is because too many features overcomplicate the model. This leads to the model overfitting more on the training data which explains the slightly worse performance. Considering that the runtime complexity is quadratic concerning F, it can be desirable to sacrifice minor performance to gain in speed. This is why we have chosen a lower F for our ASCAD models, under the assumption that the behavior concerning the number of features is the same for both datasets.

We observe that from $N_e = 8$ onward, having a higher number of edges per node is detrimental for the performance. Since our graph generation algorithm has a fixed number of edges per node, it will inevitably add low-quality edges if the fixed number of edges is high enough. Even when disregarding the quality of the edges, this graph is relatively small (50 nodes). Having a (relatively) large number of edges would oversmooth the graph without a mechanism to discriminate between edges. We can observe this in the ablation study in Section 5.6 where the performance of the GCNN on a fully connected graph is abysmal. On the other hand, the performance of the GCAT and EdgeNet architectures (which can discriminate between edges) is not as bad when using the fully connected graph.

We have chosen to relocate the results for the hyper-parameter analysis for the number of layers L and filter order K, to the Appendix. The reason for this is that those results are fairly straightforward and are not particularly interesting for analysis.

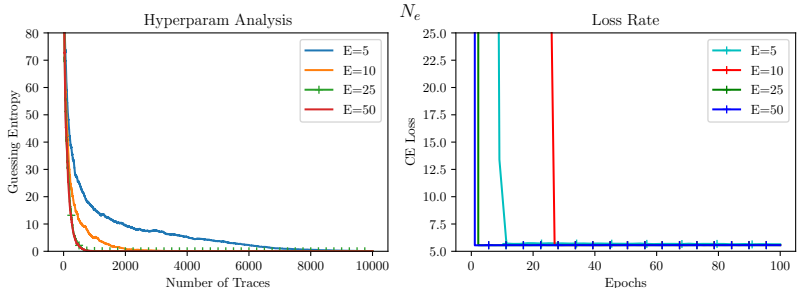

Figure 5.9: Hyper-parameter analysis on the ASCAD dataset for the number of edges per node N_e using the GCNN architecture and IV leakage model. (left) Results for different numbers of edges using only convergent instances; (right) Loss curves for different numbers of edges using only convergent instances

N_e	μ	σ	min	max	% conv
5	1599.0	2294.3	242.0	6157.0	45.5%
10	555.0	526.1	213.0	1464.0	33.3%
25	284.0	20.0	264.0	304.0	16.7%
50	272.5	22.2	238.0	298.0	40.0%
100	-	-	-	-	0.0%

Table 5.10: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are not included.

When we look at Figure 5.9, it seems that a higher number of edges results in faster convergence. That is until a certain limit as having too many edges causes the graph not to converge at all. When we look at Table 5.10, we see some interesting things. First, both the variance and the mean tend to decrease as the number of edges increases. while the percentage of convergent instances also decreases. This pattern breaks for $N_e = 50$: the variance gets slightly more, but the rate of convergent runs is a lot higher than $N_e = 25$. So when we compare the results of $N_e = 50$ with $N_e = 5$, $N_e = 50$ has a better performance at the cost of having a lower rate of successful convergences.
When we look at the losses for only the instances which converge in Figure 5.9, we see something interesting. All the losses seem to converge to the same value again. Furthermore, the loss for $N_e = 50$ jumps to this value almost instantly, similar to $N_e = 25$. Note that this behavior is very similar to the behavior shown by the loss curve of the random graph for the IV leakage model (Figure 5.7). Consider that we generate our graph by calculating the correlation between each pair of nodes and take the n closest ones as neighbors. If n is sufficiently large, then it is inevitable that some of the selected neighbors have a low correlation. So in that case, the neighbors might as well be selected randomly. So it is likely that a part of the edges which are generated for $N_e = 50$ are for all intents and purposes generated randomly. This explains why it shows behavior similar to the random graph.

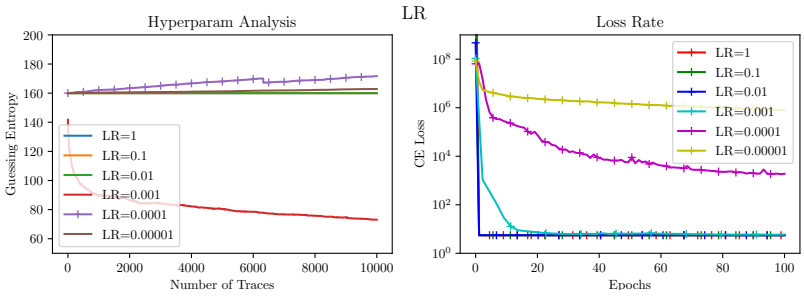

Figure 5.10: Hyper-parameter analysis on the ASCAD dataset for the learning rate.(left) Results for different learning rates ; (right) Loss curves for different learning rates

5

When we look at Figure 5.11 , we see that only for the default learning rate (10^{-3}) there is a convergent curve.[3]. When looking at the losses. we observe that for higher learning rates, the losses jump to the same value almost instantly. We see that the lower learning rates are slowly descending, but it appears that it will take a long time before they reach zero. Furthermore, there is no guarantee that using a lower learning rate will enable our model to reach a loss of zero.

Based on this experiment, we cannot say for sure whether our model gets stuck in a local optimum or that our model is finding the global optimum. It is possible that increasing the number of epochs while lowering the learning rate may result in the loss eventually converging to zero. However, lower values for the learning rate make a model more inclined to get stuck in local optima, so this seems unlikely.

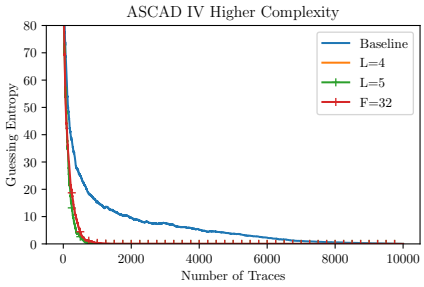

Figure 5.11: Hyper-parameter analysis on the ASCAD dataset for number of features F and number of layers L

[3]We did not include a table with numerical results since all results except the baseline were identical: no convergent instances

Setup	μ	σ	min	max	% conv
Baseline	1599.0	2294.28	242.0	6157.0	45.5%
$L = 4$	-	-	-	-	0%
$L = 5$	289.0	0.0	289.0	289.0	12%
$F = 32$	361.3	62.4	300.0	447.0	30%

Table 5.11: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are not included.

The results in Figure 5.11 and Table 5.11 show that the more complex models do not increase the number of convergent instances. In fact, adding more layers or features actually reduces the percentage of converging runs. When we ignore the outlier in the baseline, the performance of the more complex models is similar to that of our baseline. So increasing the complexity of our graph convolutional layers has little effect on the performance. So it is unlikely that the GCNN is not complex enough to solve this problem.
When working under the assumption that the MLP is the main component, it is to be expected that increasing the complexity of the GNN components would not increase the performance. However, when we increase the number of features F, we also increase the complexity of the MLP as the number of input nodes is dependent on F. And we expect that increasing the complexity of the MLP would improve the performance.

5.8. DISCUSSION

Our experiments have shown with the ablation studies that the unconnected graph has comparable performance to other graphs and that the IV leakage model outperforms the HW leakage model. Both of these findings strongly suggest that the classification MLP is the primary component of our model. Nevertheless, for the DPAv4 dataset we observe that for the GCNN and EdgeNet architectures, both the random graph and our generated graph have higher performance compared to the unconnected graph. For the ASCAD dataset, we observe that the unconnected graph has a very low rate of convergent instances than the other graphs. So we can conclude that while the graph convolutional filter layers are not the main component of our model, they improve our model's power.

We observe that for the ASCAD dataset, only the randomly generated graph consistently converges. Since this happens very consistently for multiple experiments (see Table B.15), we rule out that this happens by coincidence. At first glance, it seems weird that a randomly generated graph performs better than other graphs grounded in the structure of the data. However, we acknowledge one major difference between the random graph and the other types of graphs which we think is the leading cause: the other graphs are connected chiefly on a local level, while the random graph is connected on a global level since the edges are generated randomly. If our data functions on a global scale rather than a local scale, this would explain how the randomly generated graph performs better than the other graphs.

When trying to break the ASCAD dataset, our model could not consistently converge the guessing entropy towards zero. Increasing the complexity of our model by adding

more layers or features did not appear to resolve this problem nor was this problem caused by a (un)lucky selection of samples in the training set. The fact that increasing the number of graph convolutional layers or features did nothing for the performance is another support for our explanation that the MLP is the main component of our model. Our model is well suited to solve the desynchronization countermeasure as the graph convolution can compensate for the temporal diffusion of our data. We also expect that we can counteract higher rates of desynchronization by increasing the filter order or the number of edges per node.
While we performed an ablation study and analyzed the behavior of changing the number of edges per node, we are still not confident to say something conclusive about the quality of our generated graph. On the one hand, the ablation study shows that our graph performs better than the cyclic graph and the fully connected graph (for the GCNN architecture). On the other hand, we also see that our graph has a similar performance to the random graph, and the random graph vastly outperforms all other graphs concerning consistent convergence. We also assume that the models applied on both datasets have the same behavior, given the explicit differences like the number of features and the use of masking. This may not necessarily be true and the DPAv4 dataset could behave differently in some aspects relative to the ASCAD dataset.

It is still not perfectly clear why our model only converges sometimes for the ASCAD dataset using the IV leakage model and GCNN architecture. We think that the inconsistency is caused by the initialization of the learnable parameters. To confirm this, we would need to keep track of all learnable parameters and analyze them to see if we can find a difference between the parameters for the convergent and non-convergent instances. Another subject we need to build upon further is that the classification MLP is the main component of our model. Our confidence in this theory would be strengthened when an MLP with increased complexity (i.e., adding more layers to the MLP) would make our model converge consistently. Something else which we need to look in is why the performance of the random graph is better with regard to consistency. We have proposed earlier this section that this is likely cause by the random graphs being more globally connected compared to the other graphs. To verify this, we would need to expand the ablation studies with a graph which is randomly connected locally and another graph which is non-randomly connected globally.

6

Conclusion

In this chapter, we conclude the research we have conducted in our thesis and give pointers towards the direction of future work in this subject. In Section 6.1 we provide a brief summary of our research. In Section 6.2 we discuss how our research has answered our research questions. In Section 6.3 we discuss possible directions for future research concerning this subject based on the findings in our research. In Section 6.4 we examine the limitations we encountered in our research. In Section 6.5 we look at the broader impact of our research.

6.1. Thesis Summary

In this thesis, we proposed a method to translate a side-channel problem to a graph signal classification problem and solve this problem using graph neural networks.
In Chapter 1 we introduced the side-channel problem as well as the context of the side-channel problem. Furthermore, we presented our research questions and gave an overview of the chapters of our thesis. In Chapter 2 we elaborate on the background required for understanding our thesis. We first introduce the notation used. Next, we explain the AES cryptographic algorithm we plan to attack and techniques used in side-channel analysis. We continue with introducing concepts from graph theory that are relevant to our thesis. Finally, we examine several graph neural network architectures that are candidates for use in side-channel analysis.
In Chapter 3 we discuss related works. First, we discuss works related to side-channel analysis. Next, we discuss works on graph neural networks. Finally, we look at other applications of graph neural networks in the cyber security domain.
In Chapter 4 we describe our approach to translate the SCA problem into a graph signal processing problem. First, we explain how we are building a graph from the traces and next we describe how we use the graph in a graph neural network to solve the graph signal classification problem. Next, we show some examples of how those generated graphs look like.
In Chapter 5 we discuss our experiments. We performed several experiments to test the

performance of our model on different architectures and datasets. The results show that our model has trouble with consistently converging to a guessing entropy of zero for the ASCAD dataset. The ablation studies we've performed show that the multi-layer perceptron used for classification has a major contribution to the performance. They also show that using a random graph instead of the graph we described in Chapter 4 allows for our model to converge consistently for the IV leakage model. However, closer examination shows that the training loss does not decrease when learning a model using a random graph. Based on the results of the experiments, we conclude that the primary component of our model is the classification MLP and that the graph filter layers play a supporting role by providing feature extraction.

6.2. ANSWERS TO THE POSED RESEARCH QUESTIONS

RQ1. *How can the SCA problem be rephrased into a graph machine learning problem and which methods can we use to generate a graph based on a set of traces?*

In Chapter 4, we have rephrased the SCA problem into a graph machine learning problem by treating it as a graph signal classification problem. We treat the traces as graph signals and the corresponding time instances/features of the traces as the nodes of the graph. We generated the edges by computing the similarity between the nodes as the correlation between the node features and draw edges by taking the N_e closest nodes as neighbors for each node.

RQ2. *How does a graph neural network compare to state-of-the-art machine learning SCA techniques?*

When we compare to the results in works such as [1] or [51], our results are considerably worse for the ASCAD dataset. The main bottleneck in our results is that the guessing entropy only converges towards zero half of the time. We believe that the leading cause behind this behavior is that our model gets stuck easily in local optima depending on the random initialization of the learnable parameters of our model.

(a) *How can we apply a graph neural network to reduce the number of learnable parameters without degrading performance?*

When we look at the results of the hyperparameter experiments for the DPAv4 dataset in Section 5.7 and the Appendix, we see that we can reduce the number of features F, filter order K, and the number of layers L to some degree without an enormous impact on the performance. Since the hyperparameters mentioned above define the number of learnable parameters, reducing these hyperparameters would reduce the number of learnable parameters without (considerably) degrading performance. However, these experiments were not performed for the ASCAD dataset as those experiments would take quite some time. Furthermore, the performance on the ASCAD dataset was already not great, so we did not see the point in degrading that performance even further. The results in Section 5.7suggest that increasing the number of

hyperparameters does not influence the performance, so the reverse is also likely the case for the ASCAD dataset. There is also the problem that the classification MLP heavily contributes to the number of learnable parameters when using the IV leakage model. This ensures a hard bottom limit of the number of learnable parameters in our model. This limitation is less present when using the HW leakage model, but the results for that model were considerably worse.

(b) *What is the influence of countermeasures on graph neural networks performance?*

For the DPAv4 dataset, we can clearly see that our model finds it far more challenging to work with the masked version than the unmasked one as is shown by Figures 5.1 and 5.3. While we did not test this for the ASCAD dataset, it seems like a safe assumption that this is the case for the ASCAD dataset as well. When we consider the desynchronization countermeasure, we can see in Figure 5.3 that our model has a bit more difficulty with the desynchronized data. However, the difference in performance is less compared to the masking countermeasure. This is since we use a convolutional graph filter in our architecture on a graph with a connective pattern resembling the traces' temporal connection. So our model can somewhat compensate for the desynchronization by convolving over the entire neighborhood.

(c) *How does a graph neural network perform on different leakage models?*

All the experiments show that the IV leakage model performs better compared to the HW leakage model. The ablation studies suggest that the classification MLP has a considerable contribution towards the performance of the model. Using the IV as the leakage model means a higher number of classes (256) than the number of classes when using the HW leakage model (9). Since the complexity of the classification MLP is dependent on the number of classes, this means that using the IV leakage model would result in a more complex classification MLP. So we attribute the difference in performance to the IV model having a more complex classification MLP.

(d) *Which graph neural network architectures are (most) suitable to solve the graph machine learning problem?*

For the DPAv4 dataset, we can conclude that the GCAT is in general the worst-performing architecture. It appears that the attention mechanism is not very well suited to our problem with the data and graph we use. The main problem is that the attention mechanism ignores many other edges which are essential for our model. The GCNN and EdgeNet architectures are similar in performance, but the GCNN has a far lower time and parameter complexity than the EdgeNet.
The results for the ASCAD dataset show that only the GCNN delivers good performance. Since the GCAT also performed worse on the more straightforward dataset, it is not surprising that it performs mediocre on the ASCAD

dataset. That the EdgeNet architecture performs poorly is likely because it has too many learnable parameters, making it difficult to find a good solution.

6.3. FUTURE WORK

As far as we are aware, this is the first work that applies graph neural to a side-channel analysis problem. As such, there are multiple directions for future research. In this section, we discuss several subjects to focus on for future research.

6.3.1. USING A DIFFERENT GRAPH GENERATION METHOD

In this thesis, we have only used a single method to generate a graph. Although we have suggested some variations of this generation method, we have not performed experiments to compare the performance of different graph generation methods. Since the ablation studies show the difference in performance for graphs generated using our method and a randomly generated graph, there could be room for improvement for a more sophisticated graph generation method. In the ablation study for the EdgeNet architecture, the EdgeNet architecture can learn graphs when given a fully connected graph as input. So it would also be an option to have a learnable graph generation method based on the edge varying filter. The work in [47] shows another interesting method to achieve this by generating a new graph for each layer based on the features of the previous layer. We expect that using a different graph generation method to get a higher quality graph could help to increase the performance of the graph convolutional filters and reduce the importance of the classification MLP.

6.3.2. COMBINING GNN WITH EXISTING SCA ARCHITECTURES

We have proposed that the main component of our model is the classification MLP at the end and that the graph convolution filter layers(GCFL) play a supporting role, which can be considered as feature extractors. If we consider a GCFL with the same number of input features as output features, it would require minor modification to an existing framework to add a graph convolution filter layer. Furthermore, this GCFL would have only $K+1$ learnable parameters, so it has a minimal influence on the total complexity of the given framework. Therefore, we think that it would be interesting to add a GCFL to existing SCA architectures and see if this would increase the performance with a minimal increase of learnable parameters. Our graph generation method is agnostic to the dataset, so we can easily generate graphs for a multitude of different datasets.

We have shown that the GCNN is well-suited for solving the desynchronization countermeasure as it can counteract the temporal diffusion of the measured data. Furthermore, if we know the nature of the desynchronization, we could adapt our graph to better compensate for the desynchronization. For example, a desynchronization over a larger timeframe would require a higher number of edges than

a smaller desynchronization. Therefore, we expect that adding a GCFL will help solve datasets that make use of the desynchronization countermeasure.

6.3.3. USE ANOTHER GNN PARADIGM

In our thesis, we apply graph neural networks to an SCA problem by transforming the given SCA problem into a graph signal processing problem. However, we also considered other methods which can be used to apply GNNs to SCA: In our model, the vertices in the graph represent the time instances. However, we could also have the vertices in the graph represent the individual traces. Since we want to classify (i.e., find the associated key) the traces, this would give us a node classification problem instead of a graph signal classification problem. As we already mentioned in Section 2.4, we can also use graph neural networks for node classification. The main challenge for this method would be to generate the graph. For node classification, the underlying assumption is that nodes tend to have the same class as their neighbors. This means that the graph generation method should reflect this to get good results. This approach is explained in more detail in the Appendix C.1.

6.4. LIMITATIONS

The main limitations in our research result from the fact that this is the first work using graph neural networks for side-channel analysis. The first limitation that we identified is that it is difficult to assess whether a generated graph is suitable for our problem. This limitation becomes more complex if we consider that the suitability of a graph also depends on the choice of architecture. This is illustrated in the ablation study in Section 5.6, where we see that the fully connected graph induces a terrible performance for the GCNN but works nicely with GCAT and EdgeNet. We dealt with this limitation by choosing a single graph generation method which we found reasonable and accepting that it could be a low-quality graph. This would be alleviated by using architectures with mechanisms to correct faulty edges, such as GCAT and EdgeNet.
Another limitation we encountered from our research being novel is that it was difficult to assess the results of our model. While we could compare the performance of our model to state-of-the-art, it was complicated to determine whether particular behavior was native to our model or that there was a fault in the implementation. This resulted in us having to perform additional experiments and analysis, such as examining the validation and training losses, to get more insight into our model's behavior.

6.5. BROAD IMPACT

Graph convolutional neural networks are interesting alternatives for multi-layer perceptrons or regular convolutional neural networks for side-channel analysis. While the results of our research do not show that we should replace MLPs and

CNNs with GNNs, it is interesting to consider using graph filter layers in combination with existing SCA architectures. Graph filter layers allow architectures to have a more fine-tuned convolution defined by a graph rather than just the ordering of the features.

More broadly, our method is not constrained to just side-channel analysis. Our graph generation method works for any dataset which consists of feature vectors. This allows us to convert any arbitrary machine learning problem based on a feature vector to a graph signal processing problem. While this is not suitable for all problems, there are some problems for which this approach may be promising. A good example would be any problem using time series like our original SCA problem since a time series has a natural graph representation in the corresponding cyclic graph. We can then enhance this cyclic graph to get a more fine-tuned graph representation of this time series.

REFERENCES

[1] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ascad database. *Journal of Cryptographic Engineering*, 10, 11 2019.

[2] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

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

[4] Qi Cao, Huawei Shen, Jinhua Gao, Bingzheng Wei, and Xueqi Cheng. Popularity prediction on social platforms with coupled graph neural networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 70–78, 2020.

[5] Rohatgi P Chari S., Rao J.R. Template attacks. In Paar C. Kaliski B.S., Koç .K., editor, *Cryptographic Hardware and Embedded Systems - CHES 2002. CHES 2002. Lecture Notes in Computer Science, vol 2523*. Springer Berlin Heidelberg, 2003.

[6] Anshika Chaudhary, Himangi Mittal, and Anuja Arora. Anomaly detection using graph neural networks. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 346–350. IEEE, 2019.

[7] Jean-Sébastienand Dabbous Nora Clavier, Christopheand Coron. Differential power analysis in the presence of hardware countermeasures. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 252–263, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[8] Joan Daemen and Vincent Rijmen. The block cipher rijndael. volume 1820, pages 277–284, 01 1998.

[9] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3844–3852. Curran Associates, Inc., 2016.

[10] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The World Wide Web Conference*, pages 417–426, 2019.

[11] Fernando Gama, Antonio G. Marques, Geert Leus, and Alejandro Ribeiro. Convolutional neural network architectures for signals supported on graphs. *IEEE Transactions on Signal Processing*, 67(4):1034–1049, 2019. Accepted author manuscript.

[12] Richard Gilmore, Neil Hanley, and Maire O'Neill. Neural network based attack on a masked implementation of aes. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111. IEEE, 2015.

[13] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An aes smart card implementation resistant to power analysis attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *Applied Cryptography and Network Security*, pages 239–252, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

[15] Gabriel Hospodar, Benedikt Gierlichs, Elke Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: A first study. *J. Cryptographic Engineering*, 1:293–302, 12 2011.

[16] Elvin Isufi, Fernando Gama, and Alejandro Ribeiro. Edgenets:edge varying graph neural networks, 2020.

[17] Junteng Jia and Austion R. Benson. Residual correlation in graph neural network regression. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, Aug 2020.

[18] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):148–179, May 2019.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[20] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[21] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Regular paper introduction to differential power analysis. *J. Cryptographic Engineering*, 1:5–27, 04 2011.

[22] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[23] 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 *Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064*, COSADE 2015, page 20–33, Berlin, Heidelberg, 2015. Springer-Verlag.

[24] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. pages 3–26, 12 2016.

[25] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. volume 2587, pages 343–358, 03 2003.

[26] Olivier Markowitch, Stephane Medeiros, Gianluca Bontempi, and Liran Lerman. A machine learning approach against a masked aes. volume 5, 11 2013.

[27] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

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

[29] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pages 722–729. IEEE, 2008.

[30] Yossef Oren, Mathieu Renauld, François-Xavier Standaert, and Avishai Wool. Algebraic side-channel attacks beyond the hamming weight leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 140–154. Springer, 2012.

[31] Stjepan Picek, Annelie Heuser, and Sylvain Guilley. Template attack versus bayes classifier. *Journal of Cryptographic Engineering*, 7:343–351, 2017.

[32] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):209–237, Nov. 2018.

[33] Stjepan Picek, Annelie Heuser, Alan Jovic, and Axel Legay. Climbing down the hierarchy: Hierarchical classification for machine learning side-channel attacks. In Marc Joye and Abderrahmane Nitaj, editors, *Progress in Cryptology - AFRICACRYPT 2017*, pages 61–78, Cham, 2017. Springer International Publishing.

[34] 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 *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4095–4102, 2017.

[35] Aikaterini Protogerou, Stavros Papadopoulos, Anastasios Drosou, Dimitrios Tzovaras, and Ioannis Refanidis. A graph neural network method for distributed anomaly detection in iot. *Evolving Systems*, pages 1–18, 2020.

[36] Mathieu Renauld, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Algebraic side-channel attacks on the aes: Why time also matters in dpa. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 97–111, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[37] TELECOM ParisTech SEN research group. Dpa contest (4th edition). http://www.DPAcontest.org/v4/, 2013-2014.

[38] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[39] David I. Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

[40] Wei Song, Heng Yin, Chang Liu, and Dawn Song. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 606–618, 2018.

[41] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 443–461. Springer, 2009.

[42] François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The world is not enough: Another look on second-order dpa. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 112–129, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[43] Elisabeth Oswald Stefan Mangard and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2006.

[44] Adrian Thillard, Emmanuel Prouff, and Thomas Roche. Success through confidence: Evaluating the effectiveness of a side-channel attack. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 21–36. Springer, 2013.

[45] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[46] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 740–757, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[47] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph CNN for learning on point clouds. *CoRR*, abs/1801.07829, 2018.

[48] Lichao Wu, Léo Weissbart, Marina Krček, Huimin Li, Guilherme Perin, Lejla Batina, and Stjepan Picek. On the attack evaluation and the generalization ability in profiling side-channel analysis. Cryptology ePrint Archive, Report 2020/899, 2020. https://eprint.iacr.org/2020/899.

[49] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–63. IEEE, 2019.

[50] Yuyang Ye, Hengshu Zhu, Tong Xu, Fuzhen Zhuang, Runlong Yu, and Hui Xiong. Identifying high potential talent: A neural network based dynamic social profiling approach. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 718–727. IEEE, 2019.

[51] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):1–36, Nov. 2019.

[52] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks, 2018.

[53] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 10197–10207. Curran Associates, Inc., 2019.

A

Code

The code which has been used for this thesis can be found at https://github.com/sirpandemona/Thesis

B

ADDITIONAL RESULTS

In this chapter we present the results which we were not able to fit into the body of our thesis.

B.1. COMPARISON ARCHITECTURES

In this section we give some more background information about the different architectures we use. Table B.1 shows the values for the hyperparameters we used for each setting. Table B.2 shows the number of learnable parameters for the different architectures. We chose to reduce the number of hyperparameters somewhat for the ASCAD dataset for performance related reasons. In Table B.3 we show the average runtime for each architecture.

Architecture	F	L	K	R
GCNN(DPAv4)	64	4	2	-
GCNN(ASCAD)	16	4	2	-
GCAT(DPAv4)	8	3	5	5
GCAT(ASCAD)	4	2	4	1
EdgeNet	4	2	3	-

Table B.1: Best hyperparameter settings for each architecture

Architecture	#learnable parameters
GCNN	49152
GCAT(DPAv4)	1392
GCAT(ASCAD)	208
EdgeNet(DPAv4)	30,400
EdgeNet(ASCAD)	425,600

MLP used	#Classes	#Nodes	#learnable parameters
DPAv4(HW)	9	50	$450 \times F$
DPAv4(IV)	256	50	$12800 \times F$
ASCAD (HW)	9	700	$6300 \times F$
ASCAD (IV)	256	700	$179200 \times F$

Table B.2: Number of learnable hyperparameters for each architecture

Dataset	GCNN	GCAT	EdgeNet	GC-EdgeNet
DPAv4	4 min	9 min	5.5 min	3.5 min
DPAv4 (IV)	29 min	120 min	108 min	19 min
ASCAD	13 h	13 h	16 h	11 h
ASCAD (IV)	12 h	-	-	-
ASCAD Desync	16 h	17 h	>3 d	15 h
ASCAD Desync (IV)	17 h	-	-	-

Table B.3: Runtime for each architecture per dataset

B.2. HYPERPARAMETER ANALYSIS

In this Section, we present more results of our hyperparameter analysis.

FILTER ORDER K

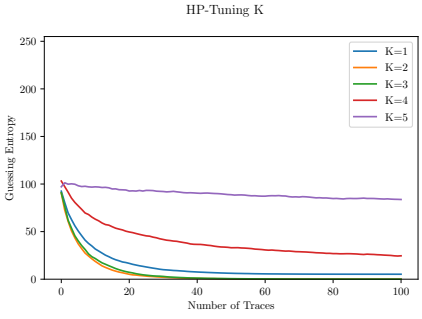

Figure B.1: Hyperparameter tuning over the filter order using the DPAv4 dataset for the GCNN Architecture

K	μ	σ	lowest	highest
1	145.7	285.1	45.0	1001.0
2	34.9	8.8	23.0	53.0
3	38.3	12.1	23.0	62.0
4	402.4	400.1	68.0	1001.0
5	950.2	103.5	703.0	1001.0

Table B.4: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

In Figure B.1 and Table B.4 we see the results of the hyperparameter analysis on the filter order K. As one can see, the performance of the model decreases when K is too high as it oversmoothes.

NUMBER OF LAYERS L

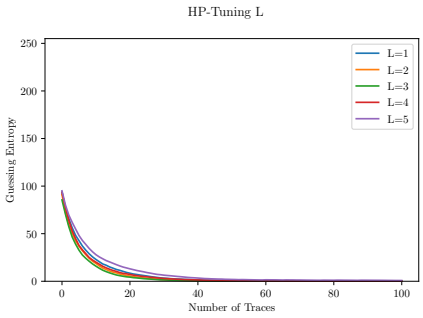

Figure B.2: Hyperparameter tuning over the number of layers using the DPAv4 dataset for the GCNN Architecture

L	μ	σ	lowest	highest
1	41.2	11.5	31.0	68.0
2	33.6	8.5	24.0	55.0
3	30.8	8.4	23.0	46.0
4	38.5	13.3	26.0	63.0
5	64.4	38.1	34.0	162.0

Table B.5: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

In Table B.5 and Figure B.2 we show the results of the hyperparamter analysis on the number of layers L. The results are all relatively close, so the number of layers does not appear to be critical for the performance.

NUMBER OF EDGES PER NODE N_e

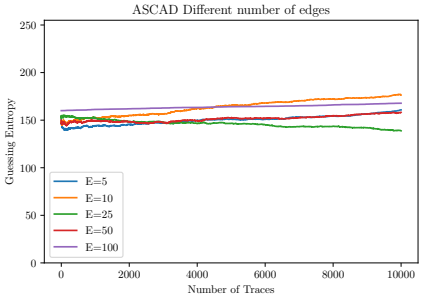

Figure B.3: Study using different number of edges for the ASCAD dataset using the GCNN architecture and HW leakage model

Number of edges per node	μ	σ	lowest	highest
5	9935.7	195.9	9348.0	10001.0
10	10001.0	0.0	10001.0	10001.0
25	10001.0	0.0	10001.0	10001.0
50	10001.0	0.0	10001.0	10001.0
100	10001.0	0.0	10001.0	10001.0

Table B.6: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

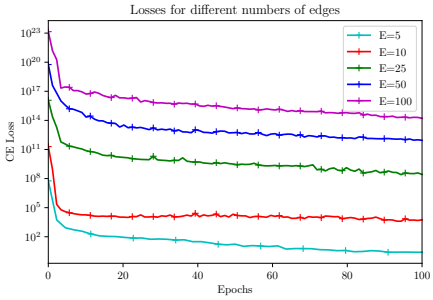

Figure B.4: Validation loss over the number of epochs during training using the GCNN architecture and HW leakage model for the ASCAD dataset

Figure B.3 and Table B.6 show the results of varying the number of edges per node for the ASCAD dataset when using the HW leakage model. Increasing the number of edges did not improve the performance of this model. Figure B.4 shows that the loss of our model increases exponentially relative to the number of edges.

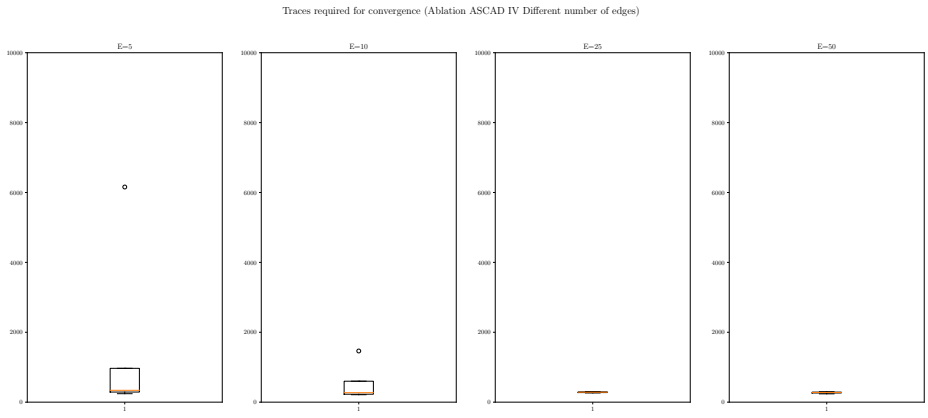

Figure B.5: Boxplot showing the distribution of the number of traces required for the guessing entropy to converge. Study using different number of edges for the ASCAD dataset using the GCNN architecture and IV leakage model where only convergent instances are used

Figure B.5 shows the distribution of number of traces each individual instance required for convergence for the hyperparameter experiment on number of edges N_e for the ASCAD dataset using the IV leakage model. We can see that a higher number of edges correlates a tighter distribution of the results.

B.3. ABLATION STUDIES

In this section we discuss ablation studies which we did not include in the main body of our thesis.

B.3.1. DPAV4

Graph type	μ	σ	lowest	highest
Baseline	22.9	3.4	17.0	28.0
Cyclic Graph	28.2	7.1	24.0	49.0
Fully Connected Graph	29.3	6.6	20.0	37.0
Random Graph	23.3	4.4	18.0	33.0
Unconnected Graph	22.7	3.2	18.0	29.0
FC Graph with 1Norm Loss of 0.01	29.3	7.96	17.0	44.0

Table B.7: Results for the ablation test for the GCAT architecture on the DPAv4 dataset.Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

Table B.7 shows the result for the ablation study for the GCAT architecture on the DPAv4 dataset. We can see that the unconnected graph has the best performance

(outperforming the baseline with a small margin), which strongly suggests that the GCAT model is unsuitable for our model.

Graph type	μ	σ	lowest	highest
Baseline	11.2	2.0	5.0	20.0
Cyclic Graph	10.1	1.20	8.0	13.0
Fully Connected Graph	6.5	1.3	4.0	12.0
Random Graph	6.3	1.4	4.0	12.0
Unconnected Graph	36.9	9.5	8.0	57.0
FC Graph with 1Norm Loss of 0.01	6.1	1.4	4.0	12.0

Table B.8: Results for the ablation test for the EdgeNet architecture on the DPAv4 dataset. Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

In Table B.8 we can see the results for the ablation study for the EdgeNet architecture on the DPAv4 dataset. We see that the unconnected graph has a considerably worse performance compared to the other graphs, which suggests that the EdgeNet model adds performance on top of the classification MLP.

B.3.2. ASCAD

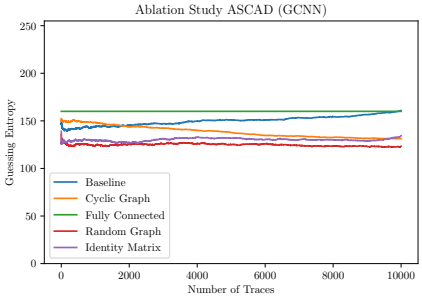

Figure B.6: Ablation study for the GCNN architecture using the ASCAD dataset using the HW leakage model

Graph type	μ	σ	lowest	highest
Baseline	9935.7	195.9	9348.0	10001.0
Cyclic Graph	10001.0	0.0	10001.0	10001.0
Fully Connected Graph	10001.0	0.0	10001.0	10001.0
Random Graph	10001.0	0.0	10001.0	10001.0
Unconnected Graph	10001.0	0.0	10001.0	10001.0

Table B.9: Results for the ablation test for the GCNN architecture on the ASCAD dataset.Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

Figure B.6 and Table B.9 show the results of the ablation study on the ASCAD dataset using the HW leakage model.

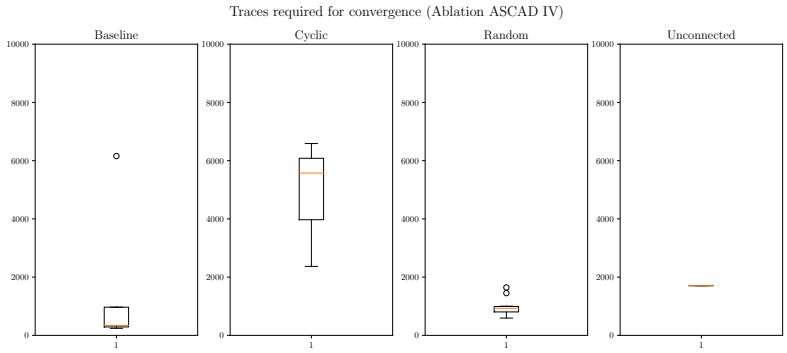

Figure B.7: Boxplot showing the distribution of the number of traces required for the guessing entropy to converge. Ablation study for the GCNN architecture using the ASCAD dataset using the IV leakage model where only convergent instances are used

Figure B.7 shows the distribution of number of traces each individual instance required for convergence for the ablation study for the ASCAD dataset using the IV leakage model. We observe that the cyclic graph has the widest distribution while the random generated graph has the tightest distribution.

B.4. LOSSES STUDY

In this Section we present the additional result for our study regarding the behavior of the validation and training losses of our model.

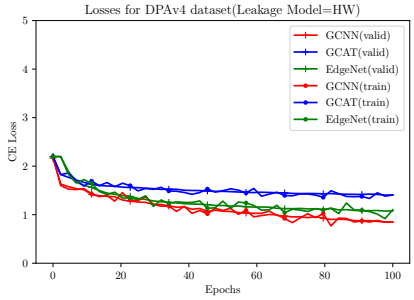

Figure B.8: Training and Validation losses over the number of epochs during training for different architectures using the DPAv4 dataset for the HW leakage model

Figure B.8 show the losses for the different architectures using the DPAv4 dataset.

The losses for each architecture are relatively close, and are in line with the performance of the corresponding architecture.

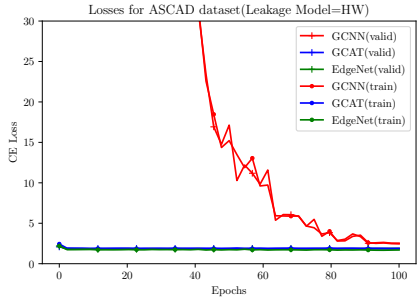

Figure B.9: Training and Validation losses over the number of epochs during training for different architectures using the DPAv4 dataset for the HW leakage model

Figure B.9 show the losses for the different architectures using the ASCAD dataset using the HW leakage model. We see that the GCAT and EdgeNet models do not appear to learn anything, the GCNN seems to be learning, but has a initial loss which is a lot higher. Furthermore, the loss for the GCNN also seems to converge towards the same value as the other architectures.

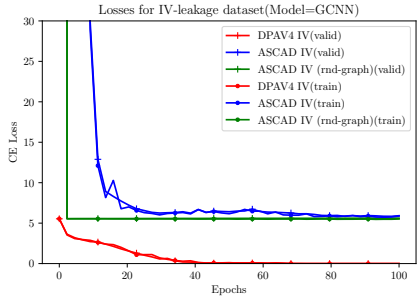

Figure B.10: Training and Validation losses over the number of epochs during training using the GCNN architecture and IV leakage model for different datasets and graph generation methods

In Figure B.10 we see the loss curves for the IV leakage model. The first thing that we see is that for the DPAv4 dataset, the losses converge to zero. As we mentioned earlier, this is the desirable behaviour of the loss curve as it implies that the model has learned the dataset in such a way that it is able to classify all the data correctly. Since the validation loss also converged, this implies that our model has not been overfit on the training data.

B.5. LEARNING RATE STUDY

In this Section we show the results for the study where we vary the learning rate of our model.

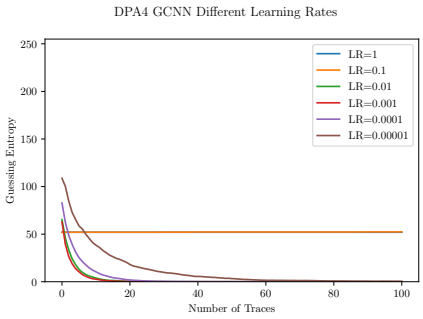

Figure B.11: Study using different learning rates for the DPAv4 dataset using the GCNN architecture and HW leakage model

Learning Rate	μ	σ	lowest	highest
10^{0}	1001.0	0.0	1001.0	1001.0
10^{-1}	1001.0	0.0	1001.0	1001.0
10^{-2}	16.3	2.7	13.0	20.0
10^{-3}	14.0	2.8	11.0	21.0
10^{-4}	23.8	2.9	21.0	31.0
10^{-5}	72.0	12.0	64	84

Table B.10: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

Table B.10 and Figure B.11 show the results for our learning rate study on the DPAv4 dataset. We observe that setting the learning rate too high will result in our model not converging.

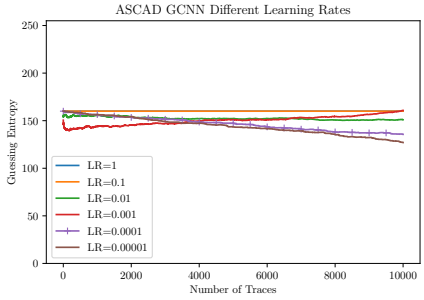

Figure B.12: Study using different learning rates for the ASCAD dataset using the GCNN architecture and HW leakage model

Learning Rate	μ	σ	lowest	highest
10^{0}	10001.0	0.0	10001.0	10001.0
10^{-1}	10001.0	0.0	10001.0	10001.0
10^{-2}	10001.0	0.0	10001.0	10001.0
10^{-3}	9935.7	195.9	9348.0	10001.0
10^{-4}	10001.0	0.0	10001.0	10001.0
10^{-5}	10001.0	0.0	10001.0	10001.0

Table B.11: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

Table B.11 and Figure B.12 show the results for our learning rate study on the ASCAD dataset using the HW leakage model. We observe that varying the learning rate does not improve our rate of convergence.

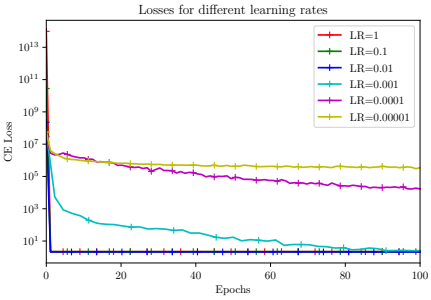

Figure B.13: Validation losses over the number of epochs during training using the GCNN architecture and HW leakage model for the ASCAD dataset.

Figure B.13 shows the losses for our learning rate study on the ASCAD dataset using the HW leakage model. We observe that a lower learning rate results in a slower

convergence of the losses, which is precisely what one would expect.

B.6. STUDY DIFFERENCE SIZE TRAINING SET

In this subsection we describe a series of experiments where we vary the number of samples used in the training set.

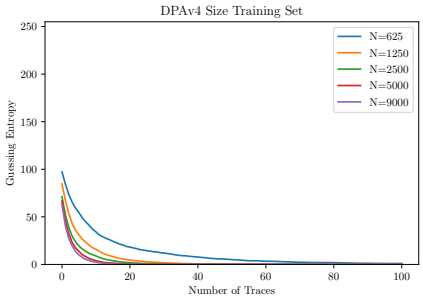

Figure B.14: Study using different sizes of training sets for the DPAv4 dataset using the GCNN architecture and HW leakage model

Size Training Set	μ	σ	lowest	highest
625	73.9	45.2	41.0	200.0
1250	33.5	8.96	25.0	56.0
2500	22.9	6.61	17.0	39.0
5000	15.8	3.92	12.0	26.0
9000	13.37	2.3	10.0	24.0

Table B.12: Table showing the mean number of traces required to reach a guessing entropy $GE < 1$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 1001.

Figure B.14 and Table B.12 both show that the performance decreases as the number of training samples decreases. Only at the lowest number of samples we used in our experiment, we notice a large difference with its predecessor.

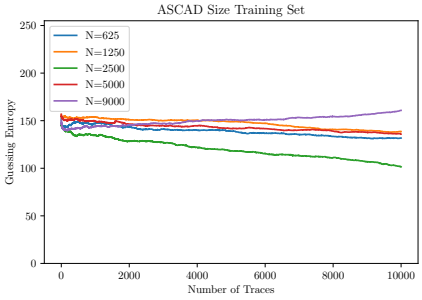

Figure B.15: Study using different sizes of training sets for the ASCAD dataset using the GCNN architecture and HW leakage model

Size Training Set	μ	σ	lowest	highest
625	10001.0	0.0	10001.0	10001.0
1250	10001.0	0.0	10001.0	10001.0
2500	10001.0	0.0	10001.0	10001.0
5000	10001.0	0.0	10001.0	10001.0
9000	9935.7	195.9	9348.0	10001.0

Table B.13: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

Table B.15 shows that when we decrease the number of training samples, our model is not longer able to find convergent instances. Since the performance of the baseline was already very poor, this is the expected behavior.

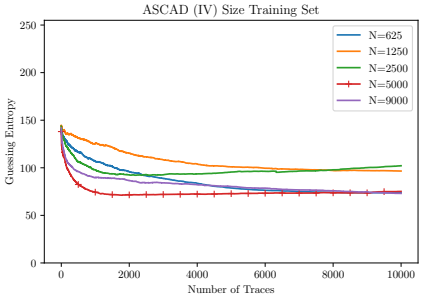

Figure B.16: Study using different sizes of training sets for the ASCAD dataset using the GCNN architecture and IV leakage model

Size Training Set	μ	σ	lowest	highest	percentage convergent runs
625	7378.43	2301.28	4783.0	10001.0	57.1%
1250	8392.43	1941.87	5031.0	10001.0	42.9%
2500	6323.57	4254.37	1095.0	10001.0	42.9%
5000	4803.14	4508.60	540.0	10001.0	57.1%
9000	6181.91	4460.40	242.0	10001.0	45.5%

Table B.14: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

Figure B.16 shows that the relation between the number of training samples and performance is less straightforward compared to the results for the DPAv4 dataset. Table B.14 shows that that there is no strong relation between the number of training samples and percentage of convergent instances.

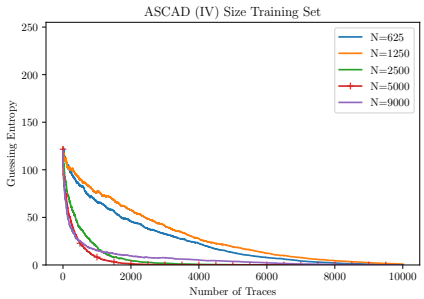

Figure B.17: Study using different sizes of training sets for the ASCAD dataset using the GCNN architecture and IV leakage model where only convergent instances are used

In Figure B.17 we ignore the instances which did not converge. This figure shows that the lower two training sizes show similar behavior and that the higher three training sizes also show similar behaviour. So it is unlikely that using an even higher number of training samples would significantly increase the performance of our model.

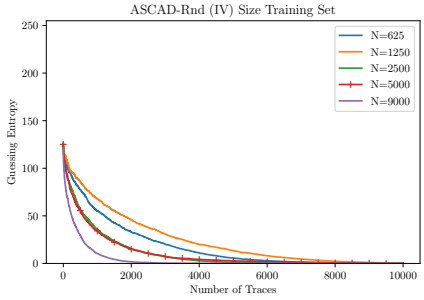

Figure B.18: Study using different sizes of training sets for the ASCAD dataset using the GCNN architecture and IV leakage model on a randomly generated graph

Size Training Set	μ	σ	lowest	highest
625	4125.3	352.06	3777.0	4900.0
1250	5473.6	658.53	4726.0	6686.0
2500	2494.27	430.19	1900.0	3308.0
5000	2513.45	1595.01	1182.0	7227.0
9000	993.4	301.566	594.0	1646.0

Table B.15: Table showing the mean number of traces required to reach a guessing entropy $GE < 10$ as well as the lower and upper bound. Runs which did not converge below this threshold are given the value 10001.

When we look at Figure B.18 and Table B.15 we observe that even for the smallest training set, all instances converge. So either our model using the random graph is really good at learning even when presented a small training set, or our model is not learning at all and something is going for which we have not found an explanation. Considering the loss curves of the random graph (as seen in Figure B.10) strongly suggests that the model is not learning, we shall consider the latter explanation. The simplest explanation would be that the random graph consistently converges purely by chance, or rather that the solution space which is explored by our model using the random graph is by happenstance very suitable for solving our problem.

C

Other Approaches

C.1. Semi-supervised

In this approach, the problem will be tackled as a semi-supervised classification problem. This means that the GNN takes a graph as input where some vertices are labeled and some vertices are unlabeled. The GNN will output the predicted labels for the unlabeled vertices. So each vertex is a trace and each label will be the key. This method transforms the original problem in a vertex classification problem. Those problems generally consist of a single graph with labeled and unlabeled vertices so the training data and test data cannot be easily separated. Therefore the test data is used by necessity during the training phase. Since an approach is semi-supervised when both labeled and unlabeled data are used during the training phase, this approach is a semi-supervised approach.
The mapping function $\mathscr{F}$ maps each trace t_i to node i with corresponding feature vector $v_i = t_i$ where $F = |v_i|$. For generating the edges we introduce a function $d(u,v)$ which measures distance/similarity between a pair of vertices (u,v). This metric is used to decide whether the graph will contain an edge $e_{u,v}$ connecting both vertices. Note that the number of vertices is constant $N = P$, but the amount of edges can vary depending on the edge selection method $N^2 \geq |E| \geq 0$. Algorithm 3 shows this method as pseudocode. In order to make this graph suitable for semi-supervised node classification, each vertex i requires a label y_i. If the corresponding trace is in the training set it will get a label corresponding with the associated key $t_i \in T_{train} : y_i = k_i$, else if the trace is in the test set it will get an empty label $t_i \in T_{test} : y_i = ""$.

The resulting graph G and label $\bar{y}$ can be used as input for the GNN. This GNN can be described as $\psi(G;\theta) = Y$ where θ represents the trainable parameters of the GNN and Y represents the predicted labels for the unlabeled vertices.

Algorithm 3 MapTraces2Nodes(T,$\bar{k}$,c)

$V, E \leftarrow \emptyset$
$(T_{train}, T_{test}) \leftarrow T$
for $i = 1 : P$ **do**
  $V \leftarrow V \cap i$
  $v_i \leftarrow t_i$
  if $t_i \in T_{train}$ **then**
    $y_i \leftarrow k_i$
  end if
  if $t_i \in T_{test}$ **then**
    $y_i \leftarrow''$
  end if
end for
$d(i, j) = \sum_{q=1}^{Q} |v_{i,q} - v_{j,q}|$
for $i = 1 : P$ **do**
  for $j = 1 : P$ **do**
    if $d(i, j) \leq c$ **then**
      $E \leftarrow E \cap (i, j)$
    end if
  end for
end for
$G \leftarrow (V, E)$
Return (G)