Author:
Green, Joseph A F

Title:
A Study of Inference-Based Attacks with Neural Network Classifiers

A Study of Inference-Based Attacks with Neural Network Classifiers

By

Joey Green

Department of Computer Science
University of Bristol

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering.

NOVEMBER 2019

Word count: 37,431

Abstract

Belief Propagation is a message-passing algorithm used to propagate information in probabilistic graphical models. In 2014 it was shown that, in theory, Belief Propagation can be applied to Side Channel Analysis through an approach in which one can recover information on the secret data of a cryptographic encryption algorithm by observing variations in power consumption or electromagnetic radiation.

In this thesis we explore the viability of such an attack in a real-world scenario and devise implementations to make the approach tractable in terms of its algorithmic and data complexity.

We explore the construction of a factor graph (a bipartite graphical representation) of the AES cryptographic algorithm, showing that not all leakage points are useful in an attack. We propose implementation improvements that significantly reduce its memory overhead. We also provide a method that guarantees convergence at the cost of a small amount of information loss. We demonstrate that a combination of these proposed methods yields a significantly improved attack in terms of memory complexity and practical runtime.

Neural networks have been applied to assist profiled side channel attacks. We contribute a new application of neural networks for inference based attacks in which we train networks for the variable nodes existing in the factor graph representation of AES. We show that popular network structures do not guarantee positive results and demonstrate that choice of performance metrics is critical in order to obtain stable results.

Our analysis indicates that there is no ‘one size fits all’ model. However, we produce a network that yields reasonable classification across all important intermediates. The results are compared to other profiling methods in two ways: through per-trace classification, and a combined approach using the Belief Propagation algorithm. We observe that the neural network assisted Belief Propagation attack outperforms classical profiling methods such as Gaussian Templating and Linear Discriminant Analysis.

Dedication and acknowledgements

This thesis took place over 4 years at the University of Bristol. I would like to thank my primary supervisor, Elisabeth Oswald, for her analytical advice and showing me the right direction. My secondary supervisor, Dan Page, helped me to hit the ground running into the glorious jungle that is Side Channel Analysis. My colleague Carolyn Whitnall provided a number of useful presentation suggestions which have made the initial chapters much easier to read. The rest of my colleagues in the 'Leaky Group' who work within the field of Side Channel Analysis were all incredibly helpful throughout my PhD life. My sponsor (GCHQ) supported my research and helped me get in contact with academics in similar fields.

My father, Stuart, proofread this thesis many times, and suggested a number of presentation and formatting corrections. The rest of my family: Sara, Alice, Tom, Sam, and Jack, were all encouraging and held my spirits high during all the moments of my PhD. My wonderful fiancée Amber continuously pushed me forward and kept me going, and provided a plethora of grammatical pointers. And finally, I would like to thank you, the reader, for taking the time to pick up this thesis and peruse through the final four years of my academic life.

This PhD is dedicated to all of you.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE: ....

TABLE OF CONTENTS

LIST OF TABLES

List of Figures

Figure **Page**

NOMENCLATURE

AES	Advanced Encryption Standard
ASCA	Algebraic Side Channel Attack
ASCAD	A Side Channel Attack Database
BP	Belief Propagation
BPA	Belief Propagation Attack
CC	Common Criteria
CMOS	Complementary Metal Oxide Semiconductor
CNN	Convolutional Neural Network
ComPred	Common Prediction (%)
CPA	Correlation Power Analysis
CPU	Central Processing Unit
DL	Deep Learning
DPA	Differential Power Analysis
DTW	Dynamic Time Warping
EE	Epsilon Exhaustion
EIS	Equal Images under different Subkeys
ELMO	Emulating Leakage for the ARM Cortex-M0
EM	Electromagnetic
GPU	Graphics Processing Unit
GTM	Ground Truth Method

HD	Hamming Distance
HW	Hamming Weight
IFG	Independent Factor Graph
IP	Internet Protocol
LDA	Linear Discriminant Analysis
LFG	Large Factor Graph
MLP	Multi Layer Perceptron
NN	Neural Network
PCB	Printed Circuit Board
PoI	Point of Interest
RF	Radio Frequency
RSA	Rivest–Shamir–Adleman, Public Key Encryption Algorithm
SASCA	Soft Analytical Side Channel Attack
SASEBO	Side Channel Attack Standard Evaluation Board
SBOX	Substitution Box
SCA	Side Channel Analysis
SCALE	Side Channel Attack Lab Exercises
SFG	Sequential Factor Graph
SMA	SubMiniature version A
SNR	Signal to Noise Ratio
SPA	Simple Power Analysis
TA	Template Attack

Chapter 1

Introduction

1.1 Research Motivation

The most ubiquitous cryptographic device in the world is the smart card, with the market exceeding 10 billion units this year (2019). With all their applications, one would expect security to be the highest concern. Unfortunately for consumers, these devices are vulnerable to exploitation via side channel attacks; an attack based on information gained from the *implementation* of some cryptographic algorithm, rather than the algorithm itself. These side channels take many forms; power consumption, electromagnetic radiation, timing information, or even sound [3] and light [4]. When data-dependent information 'leaks' through these side channels, a malicious user can use statistical analysis to recover secret data used within the cryptographic algorithm.

Both academia and industry started looking into this in detail after the Kocher et al. paper *Differential Power Analysis* [5]. Countermeasures to naive attacks have been implemented and manufactured, but the cycle continues as more sophisticated attacks are researched and demonstrated on these newly manufactured devices. The more attacks are highlighted in the research community, the harder industry has to work to secure the devices.

At the time of writing, most consumers have access to a great deal of compute power, through multi-core CPUs and powerful GPUs. Such hardware can be utilised by a malicious user to accelerate and improve the effectiveness of a side-channel based attack. Previous work has explored how one might harness the power of GPUs in effective ways [6]. The 'SASCA' attack [7] claims to be the theoretically optimal attack, by targeting every single leaking point in a cryptographic algorithm, effectively maximising the information extracted from a single trace. This is known as an *inference-based attack*, an attack that combines information from multiple leakage points to target the secret data (also referred to as a 'multivariate' attack). The reason this warrants further study is that by understanding the 'most powerful' adversary, we are able to evaluate and improve the 'worst case' security.

Deep Learning, the widely used method of image classification [8], has been applied to side channel analysis a great deal in recent literature, and is praised for its classification prowess. Again, previous work has utilised deep learning in the context of side channel analysis [9–14], but as of yet it has not been applied to an inference-based attack, and certainly not the Belief Propagation Attack as proposed by Veyrat-Charvillon et al. [7]. Belief Propagation is a well known message passing algorithm, used to propagate information around a bipartite graph known as a 'factor graph', allowing information combination from multiple sources. The two powerful tools (Belief Propagation and Deep Learning) seem to partner exceptionally well together, so this thesis aims to combine the two and create a powerful inference-based attack using neural networks as a classification tool.

1.2 Research Contributions

Our first contribution takes the form of a number of improvements to the Belief Propagation algorithm when applied to side channel analysis. As a frame of reference, to run the experiments included in this thesis (described in Chapter 4) on an Intel i7-4790 CPU @ 3.60GHz (8 cores) with 16GB RAM, the implementation of the Belief Propagation Attack as proposed by Veyrat-Charvillon et al. [7] takes 25 hours. The first improvement is a criterion that, if met, will allow the Belief Propagation to terminate early (without affecting the attack results). This is achievable by observing the quantity of information passing to the key nodes after a number of iterations. By using this improvement, we cut the expected runtime in half (12 hours using the above example). The second is a method to detect erroneous traces. This is possible in a known plaintext attack, as we compare the belief on our plaintext nodes with prior knowledge. A mismatch (over a certain threshold) signifies error, and by discarding this trace, we improve the success of our attack. The third improvement, and the most effective, is the introduction of the 'importance' metric applied to variable nodes; that is, we show that different nodes provide different quantities of information (depending on their location and leakage quality), and this can be computed efficiently. By considering how much information each node provides, we can discard the nodes that give us negligible to no information, drastically reducing the graph size from the original proposal.

We then conclude our Belief Propagation algorithm analysis by exploring some further techniques: we explore the effect of removing cycles from the factor graph, which results in guaranteed convergence, at the cost of minimal information loss. We propose novel methods to connect multiple traces together, and we present our result with the greatest success: the Independent Factor Graph method, which computes the Belief Propagation algorithm on each trace independently, combining beliefs after termination. This reduces the memory space of the attack by a considerable margin, and is the preferred method for users with limited compute power. Primarily, these improvements were developed using simulated data, but in order to test the effectiveness on real trace data, we extracted power measurements from an ARM Cortex-M0 running the `AES FURIOUS` implementation.

Our attention then turns to the classification step of the Belief Propagation attack. We compare the standard univariate templating method to the multivariate Linear Discriminant Analysis (machine learning algorithm using statistical methods to classify data), showing that the multivariate classification method improves on the Gaussian (normally distributed) univariate templates. We then go a step further, and shift our focus to Deep Learning; we aim to build Neural Networks for all intermediates in AES (that have been shown to be important in this work's previous contribution). In contrast to other work, we concentrate the tuning of our deep networks to maximise the per trace classification performance (instead of using a batch of test traces).

Our findings confirm that the 'no free lunch' theorem may have a role to play here: across all intermediates, there is *no clear winner* for the 'best' classification method (out of Gaussian univariate templates, Linear Discriminant Analysis, and Neural Networks). We find this to be true even for the same type of intermediate (e.g. the leakage corresponding to bytes during an early step of AES known as '`SubBytes`'): different algorithms perform differently across the various intermediates. Another interesting aspect of our work is related to the choice of metric used to judge the classification performance of a network. Initially we utilised the "median rank" metric (as utilised by Prouff et al. in *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* [15]) to judge the classification performance, and by doing so selecting the best network configuration (hyperparameters). When we used the "median rank" metric, the produced networks behaved in a rather arbitrary and poor manner when we trained them for different intermediates. We switched to using the "median probability" as a measure and re-discovered the best hyperparameters for a network learning the leakage related to SubBytes: this particular network configuration was able to learn the leakage for all other intermediate values very efficiently. Thus our conclusion is that although neural networks aren't always the best classifier for all the intermediates in AES, they do provide the best success when used in combination with the Belief Propagation Attack.

1.3 Thesis Outline

Chapter 1 serves as the **Introduction** chapter, complete with the motivation of this thesis, the contributions, and the outline you are currently reading.

Chapter 2 covers the **Preliminaries** required to understand the contributions made in this thesis. This includes an introduction to Side Channel Analysis, an overview of the Belief Propagation algorithm, and an introduction to Deep Learning as a classification technique.

Chapter 3 contains the **Related Work** section, providing in-depth analysis of the literature used during the development of this thesis. This is split up into sections according to content, covering papers on general side channel analysis, Belief Propagation when specifically applied to side channel analysis, Deep Learning, and an assortment of miscellaneous papers that do not comfortably fall into the previously listed categories.

Chapter 4 is the first original contribution chapter, and contains the experiments and contributions of the **Belief Propagation Algorithm**. After the introduction in Section 4.1, we first go into detail on how we implemented the Belief Propagation algorithm in Python in Section 4.2. Our Belief Propagation Attack (BPA) contributions are contained within Sections 4.4 to 4.8. Each section is split up into an introduction, experimental results, and a conclusion.

In Section 4.4 we propose an additional termination criterion to improve the practical efficiency of the Belief Propagation Algorithm. In Section 4.5 we propose a method to detect erroneous traces during the Belief Propagation Attack, and to discard these traces before they detrimentally affect the attack success. Section 4.6 studies the different methods of connecting multiple traces together, and proposes two novel techniques that reduce the memory complexity required in the original proposal of the algorithm by Veyrat-Charvillon et al. [7]. Section 4.7 studies the effect of removing nodes from the factor graph representation, the effect of which has not previously been studied in the context of side channel analysis. One of the greatest contributions made in this thesis is the introduction of the 'importance' metric, from which we can identify nodes that can be safely excluded from the analysis in order to significantly reduce computational complexity without adversely affecting the outcome. We make suggestions for when and how to remove certain nodes, which follows into Section 4.8 where we show that by removing cycles in the factor graph, we can guarantee convergence. Convergence is ideal in Belief Propagation, as it guarantees full information propagation around the factor graph, and prevents the 'chaotic' information fluctuation (see *'Evidence of chaos in the Belief Propagation for LDPC codes'* [16]). We explore what convergence means in the context of side channel analysis, and provide experimental results comparing the effectiveness of a Belief Propagation attack on different graph sizes.

Chapter 5 is our second original contribution chapter, and serves as a description of how to attack **Real Data**. After the introduction in Section 5.1, we describe the practical setup in Section 5.2. This includes all peripherals we used, and the specifications of each. Using this setup, we extract traces from the target device, which we parse using our own code; this is described in Section 5.3. Our univariate templating method is described in Section 5.3.7, along with implementation details.

We compare the trace data extracted from the real device to the simulated data using ELMO (the leakage simulation tool [17]) in Section 5.4, complete with comparison figures. Section 5.5 introduces Linear Discriminant Analysis (LDA), a multivariate classification tool which improves on the standard univariate templating approach. We find the optimal power value window for the LDA classifiers, and then we compare a BP attack using LDA to the Gaussian templates, showing the improvement that multivariate templating provides.

Chapter 6 is the final original contribution chapter of this thesis, and covers all work undertaken in the area of **Neural Networks**. We introduce neural networks in the context of side channel analysis in Section 6.1, along with the introduction of the 'No Free Lunch' theorem. Implementation details are contained within Section 6.2, and we additionally refer to the ASCAD GitHub page [18] for further implementation details.

The starting point for our work on neural networks was the paper titled *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* [15]. We provide an in-depth analysis of this paper in Section 6.3, focusing on how the authors constructed their networks, and how well the networks fared when classifying our own data (referred to as the M0 data). After concluding that we must construct our own network in a similar fashion, we experiment with different hyperparameter values in Section 6.4. This section is a compilation of separate experiments focusing on one hyperparameter at a time; we describe what the hyperparameter controls, which values we chose to test, which value was chosen as the best, and a conclusion of our observations.

The results in this section highlighted some shortcomings in the pre-existing measures of effectiveness. This prompted us to re-evaluate this metric in Section 6.5, and ultimately develop a new metric: per trace intermediate classification probability. We compare the old metric to the new metric and provide figures to show the large improvement we achieve on intermediate classification using our new superior metric. We then train networks for all intermediates and mount a Belief Propagation attack using them as classifiers in Section 6.6. We conclude with a plot showing a successful neural network assisted Belief Propagation attack that outperforms all of our previous attacks.

Chapter 7 contains the **Concluding Remarks**. We draw on the conclusions made in the original contribution chapters, listing the proposed improvements along with their respective performance benefits. We then provide an assessment of contributions, relative to related work in the field of Side Channel Analysis. We end the thesis by proposing novel areas of future work that would expand on the work done within this thesis.

1.4 Publications

A Systematic Study of the Impact of Graphical Models on Inference-Based Attacks on AES, Long Paper [19], CARDIS 2018 version [2]

Not a Free Lunch but a Cheap Lunch: Experimental Results for Training Many Neural Nets, submitted to CT-RSA 20$^{\text{th}}$ September 2019

In both submissions I was responsible for writing the code, carrying out the experiments, analysing the results, and contributing to the text.

CHAPTER 2

PRELIMINARIES

The preliminaries chapter serves to introduce the core ideas and concepts behind this thesis. The application of these concepts in related work will be described in Chapter 3.

2.1 The ARM Cortex-M0

The microprocessors used most widely in commercial smart card readers belong to the ARM Cortex family. The ARM Cortex processor family is split up into categories: the Cortex-A CPUs, which are built for performance at the cost of a large amount of power; the Cortex-R CPUs, which are built for reliability and resilience for 'mission-critical' performance; and the Cortex-M CPUs, built for energy efficient embedded devices. The ARM Cortex-M0 processor is the smallest processor of the ARM Cortex-M group, and indeed the entire ARM Cortex processor family.

It uses a 32-bit von Neumann architecture with a three stage pipeline, and implements the ARMv6-M Thumb instruction set. This includes the majority of the standard 16-bit Thumb instruction set, as well as a number of 32-bit Thumb2 instructions. We use the ARM Cortex-M0 as our target CPU throughout this thesis due to its widespread commercial use, running an 8-bit implementation of AES.

2.2 The Advanced Encryption Standard

The Advanced Encryption Standard (AES), previously known as Rijndael [20], is a symmetric key encryption algorithm. All experiments used in the content chapters of this thesis target AES as the cryptographic encryption algorithm; specifically, AES-128 (using a 128 bit key).

AES-128 takes a 128 bit key and a 128 bit plaintext as input (often the key is 'hardcoded' into the target device, so the only input required is the plaintext). The key and plaintext are modelled as a square block of 16 bytes, ordered as shown in Figure 2.1

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 2.1: 128 bits represented as a block of 16 numbered bytes

The following provides a high-level overview of the AES algorithm, accompanied by Figure 2.2:

1. `KeyExpansion`, where the 10 'round keys' are derived using Rijndael's key schedule (a series of operations using the initial 16 key bytes); these 'round keys' (each a 16 byte block) will be used in the `AddRoundKey` steps

2. Initial `AddRoundKey` (`AK`), where the plaintexts bytes are `XOR`d with the first round key bytes

3. `SubBytes` (`SB`), byte-wise substitution of the current state bytes, often using a lookup table hardcoded on the device known as a Substitution Box (`SBOX` S)

4. `ShiftRows` (`SR`), transposition of rows (not a focus point of this thesis as, in practice, can be combined with the following step)

5. `MixColumns` (`MC`), a linear mixing operation that operates on the four bytes in each column of the state bytes

6. `AddRoundKey`, as before, but using the next round key

7. Repeat from the SubBytes step for 8 further rounds (9 rounds total)

8. Final round of `SubBytes`, `ShiftRows`, and `AddRoundKey` (no `MixColumns`)

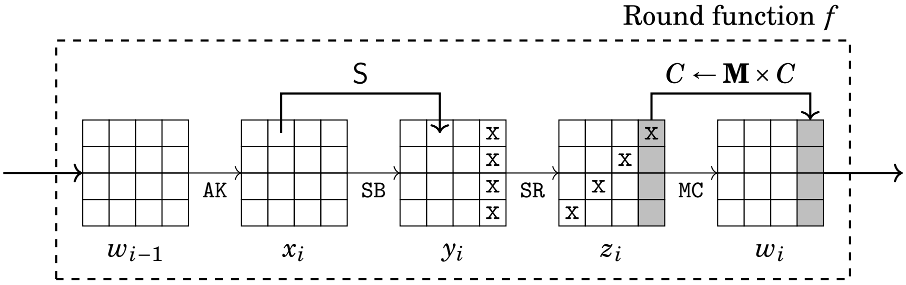


Figure 2.2: A diagram of one round of AES, where w_i is the state byte block in AES round i (w_0 is the plaintext input, and w_{10} is the ciphertext output)

The output (final sixteen state bytes) is referred to as the *ciphertext*. The implementation we use in this thesis is named `AES FURIOUS` [21], designed for fast AES implementation on AVR microcontrollers.

2.3 Introduction to Side Channel Analysis

Side Channel Analysis (SCA) targets information gained from the *implementation* of some cryptographic algorithm, rather than the algorithm itself. Side channels can take many forms; for instance, in 1996, Paul Kocher published a paper with the title *'Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems'* [22]. The paper introduced a method of exploiting the implementation of a cryptographic algorithm through observing the *time it takes* to perform the encryption step. Algorithms that change their flow of execution depending on the value of secret data are vulnerable to this attack. For example, observe Algorithm 1.

Algorithm 1: Foobar encryption, sets all bits to 0

```
   Input: A string of bits $B = \{b_0, b_1, \ldots, b_n\}$, $b_i \in \{0,1\}$
   Output: An 'encrypted' string of bits $C = \{c_0, c_1, \ldots, c_n\}$, $c_i \in \{0,1\}$
1  for $i \leftarrow 0$ to $n$ do
2      $c_i \leftarrow b_i$
3      if $c_i == 1$ then
4          $c_i \leftarrow 0$
5      end
6  end
7  return $C$
```

This 'encryption' algorithm will most likely remain unused in a practical scenario, as it will always return a string of 0's. However, it perfectly captures the timing side channel; on line 3, the algorithm uses the `if` statement, checking the value of a 'secret' bit. If the secret bit is a 1, an extra line of code is run; if the secret bit is a 0, this line is ignored. As executing an instruction takes a non-zero amount of time, if we run this encryption algorithm on two different inputs (one all 0's, one all 1's) we would observe the execution time on the latter input being longer than the former. The publication of the Kocher paper revolutionised applied cryptography and became known as the first paper highlighting the power of Side Channel Attacks.

2.3.1 Power Analysis

As mentioned, the timing side channel is only exploitable when the flow of execution is data dependent. By analysing the extracted timing data, the adversary can infer the existence of data-dependent 'leakage' (exploitable information), and mount an attack appropriately. The timing side channel shown in Algorithm 1 can be removed by replacing the 'if' statements with some constant-time alternative. Unfortunately for hardware manufacturers, there is also another vulnerability in modern devices.

Current devices used for cryptographic purposes are implemented with CMOS transistors, which make up the semiconductor logic gates used to perform the processing. When a charge flows across a transistor gate, a certain amount of power is consumed (additionally producing a certain amount of electromagnetic radiation). An adversary is able to measure the amount of power used in a circuit by implanting a resistor in series (or in some cases, through non-invasive means, such as holding an electromagnetic radiation probe over the target device [23]). The current is computed by dividing the voltage difference across the resistor with the resistance in the circuit.

The research contained in this thesis is based on exploiting the fact that the power consumption of a cryptographic device varies over time based upon the data that is processed by the algorithm implemented in that device. If the algorithm and its implementation is known to the attacker, then there is the possibility of inferring data by analysing the power consumption over time. This is the key premise of this research.

2.3.2 Simple Power Analysis

Simple Power Analysis is named thus due to the simple nature of the attack phase. The technique involves collecting a *power trace* from the target device; this is a collection of the power consumption values taken from the target device whilst a cryptographic algorithm is running.

When the target device has little noise (interference that can skew the power consumption samples), it may be possible to distinguish the exact operations occurring within the trace. For example, if we were to observe a power trace taken whilst running Algorithm 1, we would see an extra computation step taken when the current bit is set to 1 (shown in the `if` statement). By noting when this observation occurs and when it does not, it may be possible to ‘read’ the bits of the secret just by looking at a single power trace.

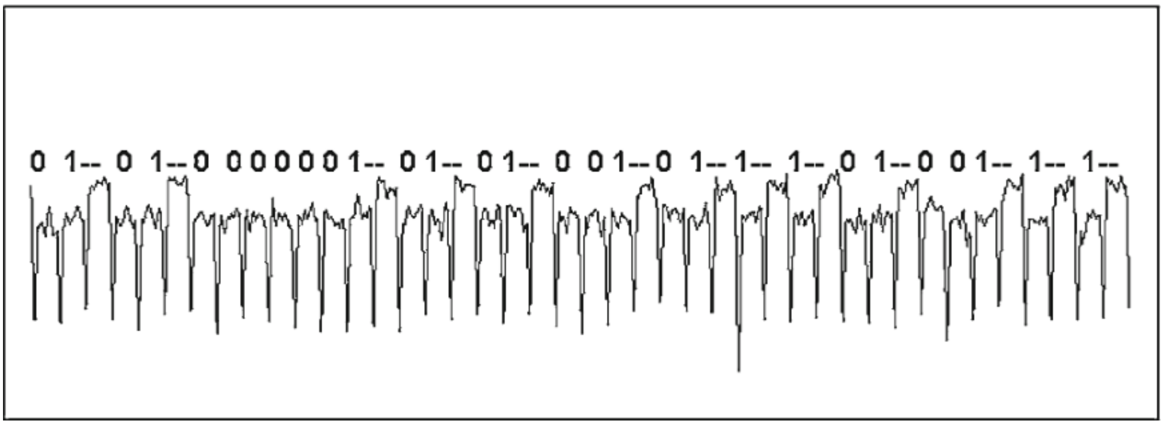


Figure 2.3: An example of a power trace with distinguishable bits, taken from *'Introduction to differential power analysis'* [1]

For example, Figure 2.3 shows a section of a power trace taken whilst the target device was running RSA (a public key encryption algorithm). Part of the RSA algorithm requires modular exponentiation, a computationally expensive operation. The method to perform this operation in the target device is known as 'square and multiply', where bits are computed left-to-right. Multiplications only occur when the exponent bit is 1, and multiplications require more power than squares. In the trace, multiplications are shown by a small peak followed by a large peak, whereas squares are simply one low peak. Knowing this, we can recover the secret bit quite easily, marking all small-large progressions as bit 1, and all other low peaks as bit 0.

2.3.3 Differential Power Analysis

Most modern devices contain a great deal of noise in their power traces, making it difficult to identify any kind of information by looking at a single trace, rendering Simple Power Analysis ineffective. In 1999, Paul Kocher published a paper on applying statistical inference to power analysis [24]. He named the technique 'Differential Power Analysis', or *DPA* for short. This method is commonly used today, and is the basis of the research in this thesis.

The idea is to generate a large number of power traces, using the same fixed key but different plaintexts, at a fixed moment of time. A typical DPA attack consists of five steps:

1. **Choose a Target Intermediate Result**. This should be a function $f(d,k)$ where d is a known data value (the plaintext) and k is a small part of the secret data (key).

2. **Measure the Power Consumption**, which essentially means we generate a number of traces, each with a plaintext generated arbitrarily. We store the plaintexts in vector $\mathbf{d} = (d_0, \ldots, d_{T-1})$, where T is the number of traces generated (one plaintext per trace). Each trace $\mathbf{t}_i = (t_i^0, \ldots, t_i^{S-1})$ corresponds to the i^{th} encryption run using plaintext d_i, where S is

the number of sample points per trace. We can then construct our matrix of traces **T** of size $T \times S$.

3. **Calculate Hypothetical Intermediate Values**. We produce a vector containing every possible value of k, named $\mathbf{k} = (k_0, \ldots, k_{K-1})$. For example, if k were a byte, this would be the vector $(0, \ldots, 255)$. Now we have vectors **d** (plaintext values) and **k** (hypothetical key values), we can produce a matrix of hypothetical intermediate values **V**, where $v_{i,j} = f(d_i, k_j)$ for i in 0 to $T-1$ and j in 0 to $K-1$.

4. **Map the Intermediate Values to the Power Consumption Values**. We now simulate the power consumption of the device to map the hypothetical intermediate values **V** to hypothetical power consumption values **H**. This can be done using a number of techniques, but the most common is using the Hamming Weight Leakage model; we therefore map **V** to **H** where $h_{i,j} = \mathrm{HW}(v_{i,j})$.

5. **Compare the Hypothetical Power Consumption Values with the Power Traces**. Finally, each column $\mathbf{h}_i$ of **H** (hypothetical power values) is compared against each column $\mathbf{t}_j$ of matrix **T** (real power values). All sample points s in matrix T are considered separately, making this step computationally expensive for large matrices. The result of this step is matrix **R** of size $K \times S$. The indices of the greatest values of matrix **R** represent the value of the key used in the target device.

For further details, see *'Power Analysis Attacks: Revealing the Secrets of Smart Cards'* [25]. In this thesis, we use the terms 'power value' and 'leakage value' interchangeably: both meaning the power consumption measurement taken from the device at a specific time point.

2.3.4 Point of Interest Detection

One of the benefits of Differential Power Analysis is that we are not required to know any algorithmic features of our target implementation: in other words, we do not need to know exactly which sample in the trace contains the leakage we wish to exploit. An exploitable sample is known as a *Point of Interest*. Step 5 of DPA compares the hypothetical power consumption values with every single column (sample) in the power trace, covering all possible leakage points.

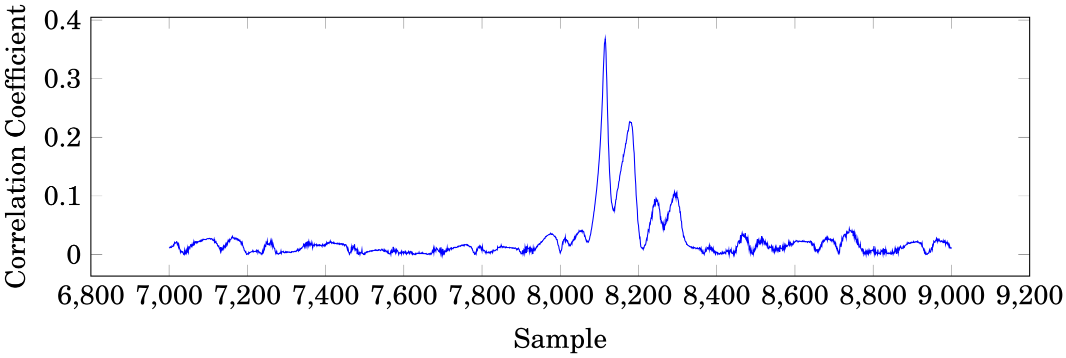


Figure 2.4: Plot showing the Correlation Coefficients for the first intermediate output in the `SubBytes` step in an implementation of `AES`

We now consider more powerful variants of DPA, that profile characteristics of the target device. The caveat is that these methods are too computationally expensive to be applied to every sample in the trace. We must first locate the Point(s) of Interest within the trace before mounting the attack.

The most straightforward way of recovering the Points of Interest is by using correlation analysis. We choose the input to the target device (plaintext and key) such that we can compute the identity (real) value of the intermediate we wish to target: this is usually an output of the `SubBytes` step. After recording the trace, we have a value-trace pair. By collecting a large number of these value-trace pairs (we found 10,000 traces to be sufficient in our work), we take each column (sample) of the traces, and correlate this to each value in the pairing. This results in a vector of correlated values of size n (number of samples per trace), known as *Correlation Coefficients*. The Point of Interest is chosen to be the sample with the highest correlated value. Figure 2.4 shows a plot of these correlation coefficients corresponding to an output of the `SubBytes` step. It is easy to spot the Point of Interest at around sample 8,100.

2.3.5 Template Attacks

Consider the following threat model: just like before, an adversary is able to produce power traces from a target device with arbitrary plaintexts, and their objective is to extract the secret key contained within the device. However, now the adversary has a 'replica' of the target device: a separate device that acts (from a leakage perspective) identically to the target device. The adversary can manipulate this replica at will, free to encrypt any message they like, providing any value for the secret key. A template is defined in this thesis as a characterisation of a univariate normal distribution consisting of two parts: a mean vector μ and standard deviation σ, represented as the pair (μ,σ). One template characterises one possible leakage value of an intermediate, so templates must be built for all possible values of the target intermediate. Employing these templates in a Differential Power Analysis attack increases the attack success, and is known as a **Template Attack**.

Template attacks consist of two phases:

1. the *offline phase*, or the *Template Building phase*, where the power leakage from the replica device is profiled and templates are created, and
2. the *online phase*, or the *Template Matching phase*, where the templates are matched with the power traces taken from the real device, resulting in a ranking of possible keys

2.3.5.1 Template Building

The adversary has full control over the replica of the target device. This allows them to target a single intermediate variable (for example, the first `SubBytes` output in the first round of `AES`), find the sample in the trace that corresponds to this intermediate variable (using the Point of Interest Detection step as described in Section 2.3.4), and collect these samples for a number of traces (providing different key and plaintext pairs for each trace). The adversary can work out the correct value of the target variable for each trace, matching it up to the corresponding leakage value, as they have access to the key plaintext pairings on the replica of the target device.

Then next step is to partition these values into a number of sets, depending on the attack strategy. One example would be to partition the values into 256 sets, one for each possible real value; as the intermediate value is a byte, the value will be between 0 and 255. Within each set, we store all the leakage values associated with this real value. Once we have 256 sets, we can compute the mean and standard deviation of each set, resulting in a (μ,σ) pair. Each pair is considered a 'template', and we require one template for every possible value. For further reading, please see [25].

2.3.5.2 Template Matching

Once all of our templates have been built, we can match these up to the leakage values extracted from traces taken from the real device. We use Bayes' rule to obtain predictions, as defined in Equation 2.1, where X is the knowledge of the trace data extracted through side channels, y is the random variable of the intermediate we are targeting, and c is a possible value of y we wish to template.

$$P(y=c|X) = \frac{P(X|y=c)P(y=c)}{P(X)} = \frac{P(X|y=c)P(y=c)}{\sum_i P(y=i|X)\cdot P(y=i)} \tag{2.1}$$

By calculating the Gaussian probability density function (defined in Equation 2.2) using a leakage value against all of our templates, we get a vector of probabilities that correspond to how likely that value was at being the correct identity of the target. We can normalise and combine these probabilities over multiple traces to form a ranking of the possible subkey.

$$P(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{2.2}$$

2.3.6 Side Channel Countermeasures

Having introduced how one might mount an attack against a cryptographic device by exploiting side channels, we now introduce countermeasures that have been developed in recent literature and implemented in commercial devices. The goal of these countermeasures is to make the power consumption of the device independent of the intermediate values in the cryptographic algorithm. These countermeasures fall into two categories: masking and hiding.

2.3.6.1 Masking

Masking randomises the intermediate values that are processed by the cryptographic algorithm. Essentially, each intermediate value v is concealed by a 'mask' m, such that $v_m = f(v,m)$.[1] This mask m is generated within the cryptographic device (and varies between encryptions), and is not known by the attacker. The masks are applied (and removed after operation) to the intermediates according to a 'masking scheme'. Different masking schemes have been proposed in recent literature [26–29], and will not be discussed in this thesis.

The advantage of masking is that it can be implemented without changing the power consumption characteristics of the device (implemented at algorithm level). The disadvantage of masking is the implementation cost, as it is expensive to add and remove masks to intermediates throughout the entire algorithm.

We do not extend our analysis to protected implementations, however we do consider an implementation that uses a more challenging leakage model than those used in recent literature [7].

2.3.6.2 Hiding

Hiding makes it difficult for an attacker to locate exploitable information within the power traces. It does this by breaking the 'link' between the processed intermediate values and the power consumption of the target device.

There are many ways to implement hiding: one could insert multiple instances of 'dummy' operations throughout the algorithm, which would not affect the state bytes within cryptographic algorithm, but would show up in the power trace and mislead the attacker. One could also 'shuffle' the sequence of execution within the algorithm (when the execution order is arranged arbitrarily). An example of this would be the `SubBytes` step in AES, where each of the 16 state bytes are passed through a lookup table and their values are substituted accordingly. The computation of these 16 state bytes can be done in any order (the lookups are all independent), so the implementer of

[1] This function is often the Boolean XOR function, but may also be the modular addition or modular multiplication depending on the cryptographic algorithm

the algorithm could shuffle the execution of these lookups to prevent an attacker from finding the timepoint of a specific intermediate. It is also possible for these two described methods to be combined.

The hiding countermeasure is popular in industry, due to the ease of implementing hiding in hardware. One advantage of hiding is that it can be implemented independently of the cryptographic algorithm running on the target device. However, there have not been many published hiding countermeasures at the software level.

We will not accommodate the hiding countermeasure in the context of this thesis.

2.4 Characteristics of Power Consumption

2.4.1 Signal-to-Noise Ratio

When we extract a number of power traces from a target device whilst it is running some cryptographic algorithm with constant input parameters, the power measurements vary over time. The fluctuations in these power traces are examples of *electronic* noise. There are multiple possibilities for the source of electronic noise: for example, it could be from the power supply of the target device, or from the clock generator (if in use), of perhaps from the combined emission of all components on the Printed Circuit Board (PCB). It is not possible to remove electronic noise completely, but it can be minimised through careful selection of hardware components.

Another form of noise is known as *switching* noise. As mentioned previously, when a logic cell switches in value, a certain amount of power is consumed, more so than if the logic cell did not switch. When we attack an implementation, we target a specific leaking operation, but at the same time, multiple other values are being leaked from neighbouring operations. From an attacker's point of view, the power consumption related to operations other than the target can be considered as noise.

The Signal-to-Noise Ratio (SNR) is a metric used to identify the ratio between the signal and the noise component of some measurement. We define SNR as we use it in this thesis in Equation 2.3.

$$\text{SNR} = \frac{\text{Var(signal)}}{\text{Var(noise)}} = \frac{\sigma_s^2}{\sigma_n^2} \tag{2.3}$$

In this thesis, we consider the Hamming Weight leakage function, where a leakage value is related to the Hamming Weight of an 8 bit value. The variance of the signal is equal to 2, which allows us to rewrite the SNR as $\frac{2}{\sigma_n^2}$. In our experiments we show simulated data using different SNRs, so the Gaussian noise is generated with respect to the Hamming Weight leakage function.

Naturally, the larger the Signal-to-Noise ratio, the easier it is to mount an attack (as the less noise there to obstruct the attack). In this thesis, we often use an SNR of 2^{-1} in examples to show cases with a relatively small amount of noise. In a real world scenario, the real device would have an SNR of around 2^{-5}, where there is much more noise.

2.4.2 Clock Jitter

A clock generator is an electronic oscillator that produces a timing signal for synchronisation within a circuit. Clock jitter is defined as the clock deviating from its ideal timing. This is usually when it is affected by some noise or interference from nearby circuitry.

In side channel analysis, the attacker acquires a time series of power measurements (traces), usually taken when the cryptographic algorithm commences. The alignment of these traces with one another is crucial to power analysis, as the attacks work by comparing specific samples (time points) across all traces. Clock jitter makes it very difficult to mount a successful power analysis attack.

Methods available to deal with jitter include: using a stable external clock generator (thus mitigating interference from nearby circuitry), using statistical techniques such as Dynamic Time Warping (as seen in *'Improving Differential Power Analysis by Elastic Alignment'* [30]), or training Neural Networks to combat the jitter through extensive training (as seen in *'Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures'* [10]).

2.4.3 Leakage Simulation

Implementing and developing side channel analysis can be difficult when the target leakage is of poor quality (excessively noisy, misaligned, etc). Instead it would be more advantageous to use simulated data as the attack target, where the developer is aware of the structure of the leakage, along with the amount of noise and misalignment (if any). Simulating leakage is not a trivial matter, however. ELMO (**E**mulating **L**eakage on an ARM Cortex-**M0**) is a tool that can simulate leakage from the ARM Cortex-M0. Excerpt from *'Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages'* by David McCann and Elisabeth Oswald and Carolyn Whitnall [31]:

"[ELMO is] a novel modelling technique that is capable of producing instruction-level power (and/or EM) models... [ELMO is] the first leakage simulator for the ARM Cortex-M0."

The creators of the tool constructed ELMO by observing the leakage of the ARM Cortex-M0 whilst running a large number of different operations in succession[2].

In order to generate leakage traces, ELMO requires the assembly source code of the target algorithm. It also needs knowledge of the inputs to instructions, so ELMO utilises the 'Thumbulator', a Thumb (16 bit ARM) instruction set simulator. By using the Thumbulator, ELMO is able to decode and execute each instruction sequentially, providing an unrolled version of the target

[2] ELMO has additional functionality to detect leakage, but for the purpose of this thesis we only use the functionality to generate leakage traces from assembly code

algorithm, along with a number of leakage traces where each instruction cycle is represented by an ELMO power value.

The ELMO power model (non-extended, as described in Section 3 of *'Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages'*) is as follows:

$$\boldsymbol{y} = \delta + [\ \boldsymbol{O_1} \mid \boldsymbol{O_2} \mid \boldsymbol{T_1} \mid \boldsymbol{T_2}\ ]\beta + \epsilon \tag{2.4}$$

where:

- $\boldsymbol{O_i} = [\ \boldsymbol{x_i}[0] \mid \boldsymbol{x_i}[1] \mid \ \ldots \ \mid \boldsymbol{x_i}[31]\ ]$ is the matrix of operand bits across bus $i = 1,2$, where x_i represents the i^{th} operand of a given instruction
- $\boldsymbol{T_i} = [\ \boldsymbol{x_i}[0] \oplus \boldsymbol{z_i}[0] \mid \ \ldots \ \mid \boldsymbol{x_i}[31] \oplus \boldsymbol{z_i}[31]\ ]$ is the matrix of bit transitions across bus $i = 1,2$, where z_i represents the i^{th} operand of the previous instruction
- δ is the scalar intercept to be estimated
- β is the vector of coefficients to be estimated
- ϵ is the vector of error terms (modelled from a Gaussian distribution with a constant variance)

By providing a compiled ARM assembly file to ELMO along with integer N (the number of traces to generate), ELMO outputs the following files:

1. The plaintexts / inputs to the target algorithm for each trace, if not chosen specifically by the user
2. The operations in the unrolled target algorithm of size l
3. N traces, each one containing l ELMO power values

The generated traces can then be used to develop and test various side channel techniques in a simulated environment.

2.5 Introduction to Belief Propagation

The BP algorithm (first described in 1987 by Judea Pearl and Thomas Verma in *'The Logic of Representing Dependencies by Directed Graphs'* [32]) is an approach designed to enable multiple discrete probabilistic distributions (each of which relating to a single component of a complex system) to be combined in order to derive a unifying probabilistic model for the whole system. It works by representing the system using a factor graph consisting of operations and operands in such a way that the discrete probabilistic distributions can be passed back and forth in the form of messages. After passing these messages (i.e. propagating the information) around the factor graph, one can compute the marginal of a variable (the probability distribution taking into account all distributions in the factor graph) by taking the product of all messages sent directly to the variable. The following section will briefly describe the main elements of Belief Propagation as they are used within the thesis. The description is a paraphrased version taken from *Information Theory, Inference and Learning Algorithms* [33].

2.5.1 Graph Structure

The Belief Propagation Algorithm in its simplest form is applied to a ‘factor graph’: an undirected bipartite graph split into variable nodes (depicted as circles) and factor nodes (depicted as rectangles). Variable nodes represent the state of a variable. The factor nodes provide details on the relation between the variable nodes.

The factor graph is built to reflect the steps of an algorithm. As an example, consider Algorithm 2.

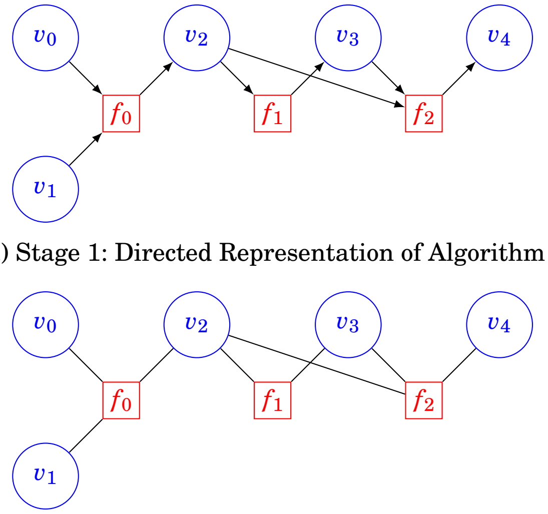
Algorithm 2: Simple Algorithm to Demonstrate Factor Graph Construction

```
Input: Two variables $v_0$ and $v_1$
Output: Output variable $v_4$
$v_2 = f_0(v_0, v_1)$
$v_3 = f_1(v_2)$
$v_4 = f_2(v_2, v_3)$
return $v_4$
```

We wish to build a factor graph representation of Algorithm 2. The factor graph construction step consists of two stages:

1. Construct an intermediate *directed graph*, such that the directed edges reflect the flow of data in the execution of Algorithm 2. The variables of Algorithm 2 ($v_0, \ldots, v_4$) are represented as variable nodes, and the functions that operate on the variables (f_0, f_1, f_2) are represented as factor nodes, as shown in Figure 2.5a. In this representation, the data always flows in one direction (left to right in this example).

2. We now use the constructed directed graph for a different purpose. Our intention is to associate statistical information with each node, and then pass this information between nodes along the edges in *all directions*, which allows us to collect the combined information on a specific variable node. To allow messages to be passed in all directions, we remove the directional dependencies in the directed graph to produce an undirected factor graph, as shown in Figure 2.5b.



(a) Stage 1: Directed Representation of Algorithm 2

(b) Stage 2: Undirected Representation of Directed Graph

Figure 2.5: The Stages of Factor Graph Construction

It is important to note here that by removing the directional dependencies from the directed graph, we may produce cycles (paths from a variable node back to itself without traversing any edge more than once). This is illustrated in Figure 2.6, where a cycle of four edges has been created (indicated by the non-dashed lines).

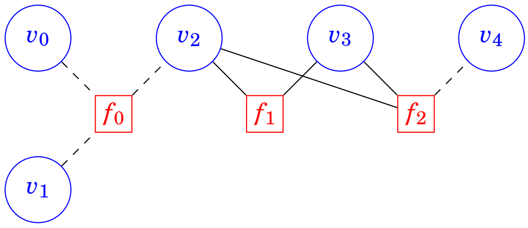


Figure 2.6: Cycle created by removing directional dependencies, as indicated by the bold lines

The following definition was taken from *'Information theory, inference and learning algorithms'* [33]. We define a function P^* of a set of N variables $\mathbf{x} \equiv \{x_n\}_{n=1}^{N}$ as a product of M factors, as in Equation 2.5.

$$P^*(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m) \tag{2.5}$$

Each factor $f_m(\mathbf{x}_m)$ is a function of a subset $\mathbf{x}_m$ of the variables that make up $\mathbf{x}$. Following our example in Figure 2.5, f_0 was originally a function of the variable subset $\{x_{v_0}, x_{v_1}\}$ in the directed representation of the algorithm, but by removing the directional dependencies, it becomes a function of the variable subset $\{x_{v_0}, x_{v_1}, x_{v_2}\}$.

We wish to be able to compute the marginal function of any variable x_n, defined in Equation 2.6.

$$Z_n(x_n) = \sum_{\{x_{n'}\},n' \neq n} P^*(\mathbf{x}) \tag{2.6}$$

This sums over all the variables associated with x_n, except for x_n itself. Following the previous example, the marginal function of variable v_0 would be

$$Z_0(v_0) = \sum_{v_1,v_2} f_0(v_0, v_1, v_2)$$

For further details, see *'Information theory, inference, and learning algorithms'* [34].

2.5.2 Constructing a factor graph for AES FURIOUS

In the context of Side Channel Analysis, we target cryptographic algorithms, such as the block cipher AES. We then use leakage information extracted from power measurements to infer information on certain variables; most notably, the key used in the encryption / decryption method. We can easily translate a cryptographic algorithm into a factor graph, and we will use AES as an example.

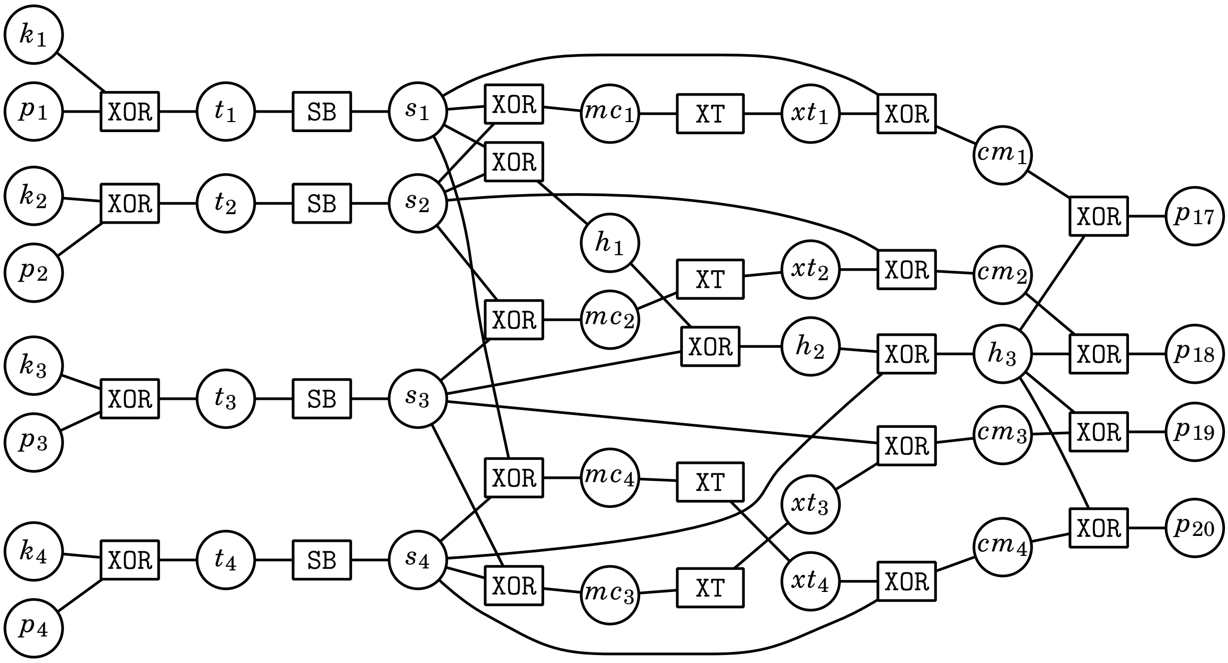


Figure 2.7: Factor Graph representation of the first column of the first round of AES, originally presented in my CARDIS publication *'A Systematic Study of the Impact of Graphical Models on Inference-Based Attacks on AES'* [2]

To start, we first construct variable nodes representing the sixteen initial plaintext and key bytes, as these are provided as input to the AES algorithm. Figure 2.7 shows the first column of the first round of AES, in which the first four key bytes and plaintext bytes are represented as the variable nodes $k_{1,\dots,4}$ and $p_{1,\dots,4}$ respectively. We then inspect the AES assembly code line by line. Each time we see an arithmetic operation (ignoring memory operations such as loading and storing) being performed, we construct a new 'factor' node representing the computed operation (e.g. XOR). We also create a new variable node (named uniquely but arbitrarily) to represent the output of this operation. This new variable node is connected to the new factor node. We also connect the input(s) to the operation, which will have previously been created using this method. In this way, we continue adding nodes to our graph, until we have a factor graph that represents the entire AES algorithm. In the AES FURIOUS implementation, the factor graph representation contains the XTIMES, XOR, and SBOX factor nodes, as shown in Figure 2.7.

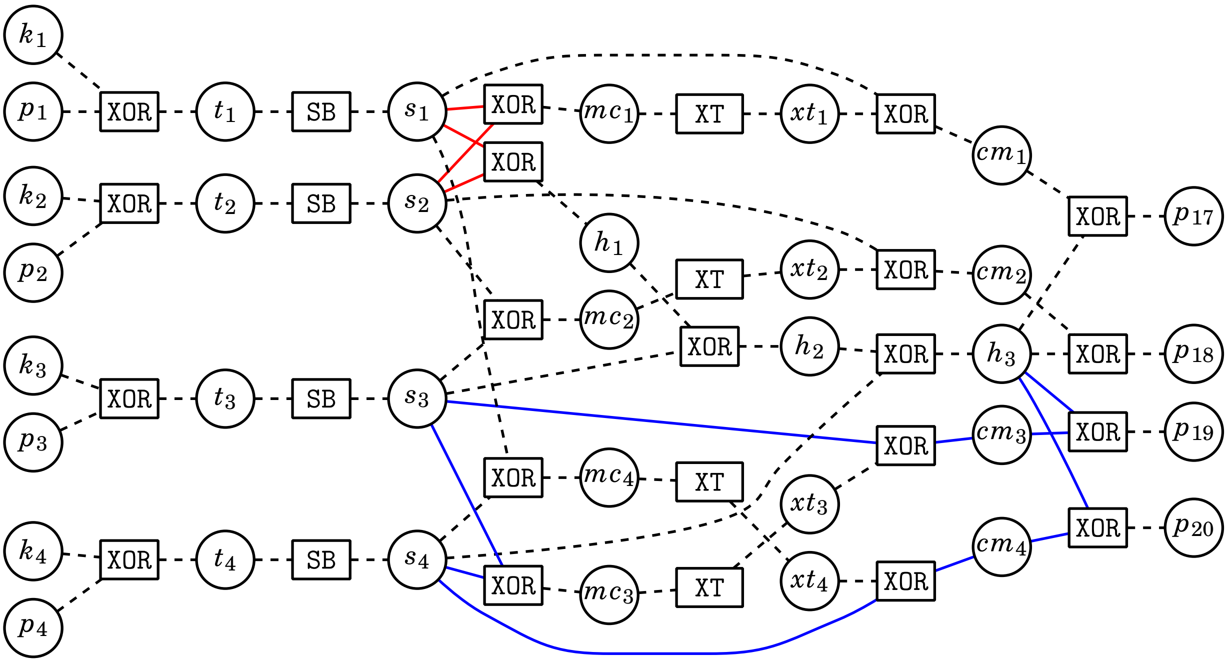


Figure 2.8: A factor graph representation of the first round of AES FURIOUS limited to one column, with two cycles highlighted: one small cycle in red, one larger cycle in blue (all other edges are dashed for visual aid). Note that Factor Graphs are inherently undirected.

It is important to note that this factor graph contains multiple cycles, as illustrated in Figure 2.8.

2.5.3 The Belief Propagation Algorithm

Each edge in the factor graph represents *two* discrete probability distributions relating to the connected variable node: one representing a message from the variable node to the factor node, and one from the factor node to the variable node. The Belief Propagation Algorithm updates these edges (over a number of BP iterations) according to a set of rules.

Following the notation by David MacKay in *"Information theory, inference and learning algorithms"* [33], we define the set of variables that the mth factor depends on $((x)_m)$ by the set of their indices $\mathcal{N}(m)$. Similarly, the set of factors in which variable n participates is defined as $\mathcal{M}(n)$. Many of the functions involve 'including all but one' variable, so we define the set $\mathcal{N}(m)$ with n excluded as $\mathcal{N}(m)\backslash n$. We also use the following shorthand $\mathbf{x}_m\backslash n$:

$$\mathbf{x}_m\backslash n \equiv \{x_{n'} : n' \in \mathcal{N}(m)\backslash n\} \tag{2.7}$$

There are two types of messages passed in the Belief Propagation algorithm:

- messages $q_{n \longrightarrow m}$ from variable nodes to factor nodes
- messages $r_{m \longrightarrow n}$ from factor nodes to variable nodes

It is important to note that the messages are probability distributions corresponding to the connected variable node: in other words, the message $q_{n \longrightarrow m}$ and the message $r_{m \longrightarrow n}$ both hold a probability distribution of variable x_n. In David MacKay's original notation of the sum-product algorithm, the *a priori* information of a variable node (leakage information taken from the trace data) is modelled as a leaf factor node connected to the variable node (as will be demonstrated in Figure 2.11. However, we opt to remove these leaf factor nodes from the factor graph representation and instead include an *initial distribution* L_n that is initialised and stored within each variable node.

The rule for updating the messages sent from variable nodes to factor nodes is defined in Equation 2.8, and Figure 2.9 is provided as a visual representation of the direction of messages (representing the AddRoundKey and SubBytes steps for one key and plaintext byte).

$$q_{n \longrightarrow m}(x_n) = L_n \cdot \prod_{m' \in \mathcal{M}(n)\backslash m} r_{m' \longrightarrow n}(x_n) \tag{2.8}$$

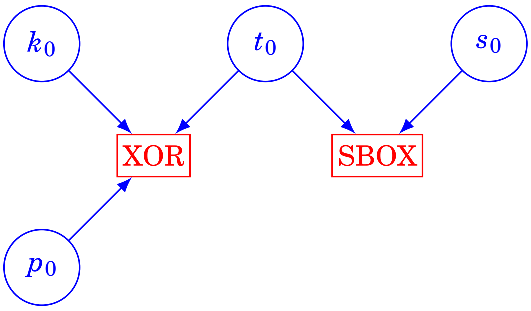


Figure 2.9: An illustration of the Variable Pass step in the Belief Propagation Algorithm

The rule for updating the messages sent from factor nodes to variable nodes is defined in Equation 2.9, and Figure 2.10 is provided as a visual representation of the direction of messages.

$$r_{m\longrightarrow n}(x_n) = \sum_{\mathbf{x}_m\setminus n}\left(f_m(\mathbf{x}_m)\prod_{n'\in\mathcal{N}(m)\setminus n} q_{m'\longrightarrow n}(x_n)\right) \tag{2.9}$$

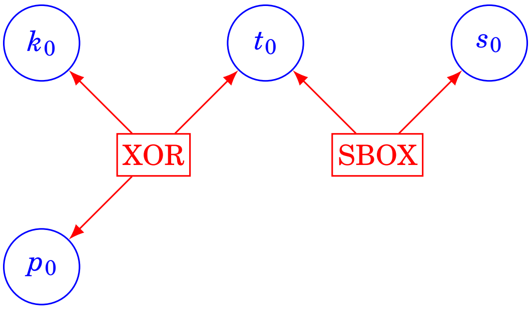


Figure 2.10: An illustration of the Factor Pass step in the Belief Propagation Algorithm

The initial set up of the graph is different for leaf variable nodes; we define it in Equation 2.10.

$$\text{For leaf variable nodes } n\text{: } q_{n\longrightarrow m}(x_n) = L_n \tag{2.10}$$

The algorithm runs for a series of iterations. Within each iteration, first all the variables will pass their messages to each of their neighbours according to equation 2.8, and once completed, the factors will pass their messages according to equation 2.9. Variable nodes (and factor nodes respectively) can perform their message passing in parallel, as the graph is bipartite, ensuring independency between message passing. Of course, all variable (factor resp.) nodes must finish passing their messages before factor (resp. variable) nodes can start to pass their messages.

As information propagates around the graph over a number of BP iterations, the edges are continuously being updated. When the factor graph is tree-like (no cycles), BP will 'converge' after a certain number of BP iterations: this is when all information has propagated fully around the graph, and the messages rest in a stable equilibrium, no longer being updated in successive BP iterations. Convergence is only guaranteed when there are no cycles in the graph.

At the end of the Belief Propagation Algorithm (after all BP iterations, regardless of whether BP has successfully converged), we compute the marginal of a variable node; this is the product of all incoming messages and the initial distribution of the variable, as defined in Equation 2.11. Figure 2.11 represents how we might compute the marginal of the k_0 node; we take the product of the message from the `XOR` node (which is the product of the messages sent to the `XOR` node from variable nodes p_0 and t_0) with the initial distribution of k_0, denoted as L_{k_0}. The collection of marginals on specific variable nodes (in our case, the initial 16 key bytes) becomes the output of the Belief Propagation algorithm.

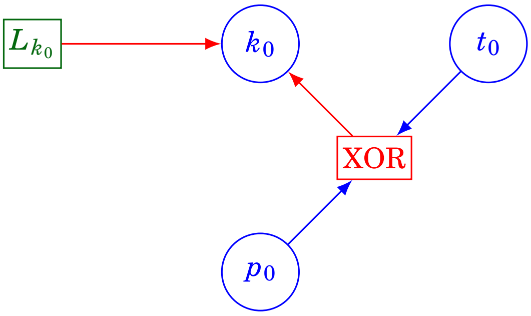


Figure 2.11: An illustration of how the Marginal is computed for a Variable Node

$$\text{marginal}(x_n) = L_n \cdot \prod_{m \in \mathcal{M}(n)} r_{m \longrightarrow n}(x_n) \tag{2.11}$$

2.5.4 A Worked Example of Belief Propagation

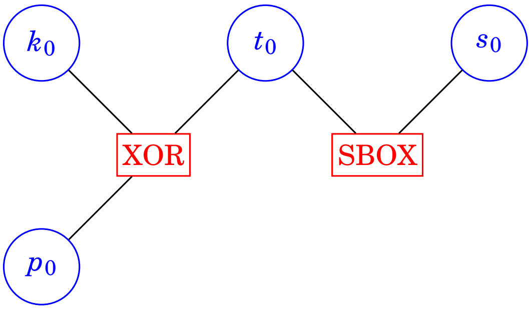


Figure 2.12: An illustration of the Factor Graph used in the worked example

We provide a worked example of the Belief Propagation Algorithm, using the factor graph shown in Figure 2.12. In this example, we play the role of the adversary attempting to recover the value of the secret key node k_0. We make three assumptions to simplify the illustration:

1. Each variable node represents a 3 bit value from 0 to 7 (as opposed to the 8 bit values assumed later in this thesis, with values ranging from 0 to 255)

2. Values leak according to the Hamming Weight leakage model (we know the Hamming Weight of all values apart from the plaintext, of which we know the correct value)

3. We define the SBOX function as $S(x) = (x-1) \bmod 8$, e.g. $S(1) = 0$ and $S(0) = 7$

Variable	Real Value	Hamming Weight	Initial Distributions
k_0	1	1	$\{0,\frac{1}{3},\frac{1}{3},0,\frac{1}{3},0,0,0\}$
p_0	2	N/A	$\{0,0,1,0,0,0,0,0\}$
t_0	3	2	$\{0,0,0,\frac{1}{3},0,\frac{1}{3},\frac{1}{3},0\}$
s_0	2	1	$\{0,\frac{1}{3},\frac{1}{3},0,\frac{1}{3},0,0,0\}$

Table 2.1: The values and initial distributions of the variable nodes in the worked BP example; note that the Hamming Weight is not applicable for the variable node p_0 as the real value is known to the adversary

Table 2.1 shows the real values used in the example, along with their leaked Hamming Weight values and resulting initial distributions. This example is intended to simulate a real world scenario: as the adversary, we do not have access to the real values (bar the plaintext node), but are able to extract the leaked Hamming Weights from a physical device. An alternative interpretation of the initial distribution of k_0 is shown in Equation 2.12.

$$P(k_0 = x) = \begin{cases} \frac{1}{3} & \text{if } x \in \{1,2,4\} \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

The variable nodes in the factor graph are populated with their respective probability distribution, in preparation for the first BP iteration.

2.5.4.1 Iteration 1

As described in Section 2.5.3, BP iterations consist of two phases: the variable pass phase, followed by the factor pass phase. The variable pass phase in the first iteration is simple: the variables pass their initial distributions to all adjacent factor nodes, as illustrated in Figure 2.9.

The factor pass phase requires the factor nodes to process incoming messages, then send the processed messages to other neighbouring variable nodes, as illustrated in Figure 2.10. For example, the XOR node must send its message to the key node k_0. This is computed by performing the XOR operation over the messages sent to the XOR node by the p_0 and t_0 variable nodes, as shown in Listing 3 (for further reference, the python code to perform this calculation will be provided in Listing 4.1).

The message from the XOR factor node to the key byte k_0 is as follows:

$$r_{\texttt{XOR}\longrightarrow k_0}(x_{k_0}) = XOR(q_{p_0\longrightarrow \texttt{XOR}}, q_{t_0\longrightarrow \texttt{XOR}}) = \{0,\frac{1}{3},0,0,\frac{1}{3},0,0,\frac{1}{3}\}$$

Once the XOR factor node has sent its message to k_0, it then sends the appropriate messages to p_0 and t_0. However, as these are propagating away from the key node (which we wish to recover), we will not include these messages in this example.

Algorithm 3: Algorithm to compute the XOR of two probability distributions $XOR(a_0,a_1)$, where the normalise() function makes the array sum to one

```
Input: Two arrays $a_0$ and $a_1$ (representing probability distributions)
Output: Output array $a_2$
$l \leftarrow$ size($a_0$)
$a_2 \leftarrow$ array of 0s of size $l$
for $i = 0; i < 2^l; i++$ do
    for $j = i; j < 2^l; j++$ do
        $a_2[i \oplus j] \leftarrow a_2[i \oplus j] + (a_0[i] \cdot a_1[j])$
    end
end
return $normalise(a_2)$
```

The SBOX node sends its message to t_0 by performing the SBOX operation on the incoming message from the s_0 node. As this is a 1-to-1 mapping (only one input) it effectively permutes the incoming message, resulting in the following message:

$$r_{\texttt{SBOX}\longrightarrow t_0}(x_{t_0}) = SBOX(q_{s_0\longrightarrow\texttt{SBOX}}) = \{0,0,\frac{1}{3},\frac{1}{3},0,\frac{1}{3},0,0\}$$

The iteration ends after all factor nodes have passed their messages to all their respective neighbouring variable nodes. At this point we are able to calculate the marginal probability of k_0, as illustrated in Figure 2.11. Equation 2.11 calculates the marginal as the product of the initial distribution of k_0 with all incoming messages to k_0 (in this example, the message $r_{\texttt{XOR}\longrightarrow k_0}(x_{k_0})$), which would result in the following probability distribution:

$$\text{marginal}(x_{k_0}) = \{0,\frac{1}{2},0,0,\frac{1}{2},0,0,0\}$$

The marginal distribution at this point can be interpreted as “all the information we know about k_0 after one iteration of BP”. We can see we have reduced the possible key space from 3 values down to 2.

2.5.4.2 Iteration 2

The variable pass phases in the second iteration onwards differ from the first, in that the (non-leaf) variables must take into account incoming messages, as shown in Equation 2.8. The only variable affected in this example is variable t_0, which must now pass the product of L_{t_0} (the initial distribution of t_0) with the incoming message from the SBOX factor node, $r_{\texttt{SBOX}\longrightarrow t_0}(x_{t_0})$. This results in the following probability distribution being sent to the XOR node:

$$q_{t_0\longrightarrow\texttt{XOR}}(x_{t_0}) = \{0,0,0,\frac{1}{2},0,\frac{1}{2},0,0\}$$

All leaf variable nodes continue to send their initial distribution in the variable pass phase, and thus can be ignored. In the factor pass phase, the XOR node now has some new information from t_0, and will send the following message to k_0:

$$r_{\texttt{XOR}\longrightarrow k_0}(x_{k_0}) = XOR(q_{p_0\longrightarrow\texttt{XOR}}, q_{t_0\longrightarrow\texttt{XOR}}) = \{0, \frac{1}{2}, 0, 0, 0, 0, 0, \frac{1}{2}\}$$

Once this message has been sent, all information has successfully propagated throughout the factor graph, and the Belief Propagation Algorithm terminates. The marginal of k_0 is computed as:

$$\text{marginal}(x_{k_0}) = \{0, 1, 0, 0, 0, 0, 0, 0\}$$

We have successfully recovered the value of k_0 as 1. This would be classified as a first order success (the exact value of the key is known).

2.5.5 Using Belief Propagation in SCA

The work presented in *'Soft Analytical Side-Channel Attacks'* [7] provides a description of how to use Belief Propagation in Side Channel Analysis. The attack is similar to a Template Attack, where the adversary has access to a replica of the target device. We will use AES as an example, as it is a widely used block cipher.

As described in Section 2.5.2, we are easily able to convert the underlying cryptographic algorithm into a factor graph. We then build profiles of the power leakage for each variable in the converted factor graph from the replica of the device. We run the cryptographic algorithm using random plaintext and key pairs, storing the power leakage information for each trace along with the key plaintext pair. We can then collect the power values for each respective variable in the graph by using correlation analysis on the power traces.

Once we have profiled all variables in the factor graph, we can then mount our attack on power traces taken from the real device. By using the same correlation analysis we extract the power values for each variable, and by using the template matching step as described in Section 2.3.5 we produce a probability distribution for each variable node. These probability distributions become the 'messages' we propagate around the graph as described in Section 2.5.3.

2.5.6 Probability Distribution Distance Metrics

As previously described, the messages propagating around the factor graph during Belief Propagation take the form of probability distributions: vectors of probabilities that sum to 1. We now define metrics that compare the 'similarity' (or distance) between two probability distributions, which will be utilised in Chapter 4. We define the probability distribution a of a random variable X in Equation 2.13, where n is the number of possible values X could be.

$$a = \{p_1, p_2, \ldots, p_n\}, \qquad p_i = P(X = i), \qquad \sum_{i=1}^{n} p_i = 1 \tag{2.13}$$

2.5.6.1 Euclidean distance

The Euclidean distance (previously known as the Pythagorean metric) is the straight line distance between two vectors (or points) in Euclidean space. For two discrete distributions p and q the Euclidean distance is defined in Equation 2.14.

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{2.14}$$

2.5.6.2 Hellinger distance

The Hellinger distance metric (introduced by Ernst Hellinger in 1909) is used in probability and statistics to quantify the similarity between two probability distributions. For discrete distributions, the Hellinger distance metric is directly related to the Euclidean distance metric (straight-line distance between two points in Euclidean space).

For two discrete distributions p and q the Hellinger distance is defined in Equation 2.15.

$$D(p,q) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i}^{n} (\sqrt{p_i} - \sqrt{q_i})^2}. \tag{2.15}$$

This produces a value between 0 and 1, where a 0 indicates the distributions p and q are identical, and a 1 is achieved when p assigns probability zero to every index in which q has assigned a positive probability (and vice versa).

2.6 Introduction to Machine Learning

Machine Learning is the concept that computer systems ‘learn’ how to perform specific tasks without using explicit instructions. They do this by using algorithms and statistical models that rely on finding patterns and inferring information from training data. Machine Learning is recognised as a subset of Artificial Intelligence. In this thesis we use Linear Discriminant Analysis as our first step into Machine Learning.

2.6.1 Linear Discriminant Analysis

Originally developed by Sir Ronald Fisher in 1936, Linear Discriminant Analysis (LDA) is a generalisation of Fisher’s linear discriminant, which is a statistical method used to find a linear combination of features that characterises multiple classes of objects. LDA is closely related to regression analysis, and is a widely used tool as a linear classifier (a tool that classifies objects based on the value of a linear combination of the characteristics). By considering data over a large window surrounding a Point of Interest, the LDA approach is an example of a Machine Learning multivariate templating method. We consider multiclass LDA, as we wish to classify each intermediate byte (256 values). In the training phase, LDA models the conditional distribution of $P(X|y = c)$ for each class c (and a vector of power values X). In the classification phase, LDA uses Bayes' rule to obtain predictions, in an identical manner to the univariate templating method as described in Section 2.3.5.2.

2.6.2 Introduction to Deep Learning

Deep Learning is a subset of Machine Learning. There is ambiguity in literature on the strict divide between Deep Learning and Machine Learning. In this thesis, we define the distinction between the two as follows: when a Machine Learning algorithm predicts something incorrectly, the human user must intervene to adjust the model manually. In Deep Learning, the model can determine whether the prediction is inaccurate on its own, and adjust itself accordingly. In other words, Deep Learning learns features and model structure from unprocessed data, and continues to self-improve without intervention. Other Machine Learning methods require more preprocessing and guidance and hit limits sooner so that, after a point, new data does not necessarily contribute to an improved model.

Neural networks are frameworks for machine learning algorithms to process complex data inputs, and are examples of Deep Learning. Neural networks consist of a connected network of simple processing nodes, where the nodes model the neurons in the human nervous system. Neural networks are used extensively in the field of image recognition, where they are able to identify images that contain specific objects. To be able to do this, they must be 'trained' by being provided with some images in which the specific object is present, along with some images where the specific object is not present. From this, the network can 'learn' key features about the images, allowing them to classify images with greater precision. Neural networks have since been applied to many different fields: voice recognition, medical diagnosis, and in our case, Side Channel Analysis.

2.6.3 How Neural Networks Work

A neural network is a collection of connected neurons, often called nodes or units. A node is where the computation happens, built to mimic the neuron within the human brain. Figure 2.13 provides a visual representation of a node. Its function is to combine inputs x_i with respect to weights w_i, which either amplify or dampen the corresponding input. The idea behind this is that some inputs might be more important than others when solving a specific task; the weights are able to sift out unimportant pieces of information. The products of the inputs and respective weights are summed within the input function, upon which an activation function is applied. This

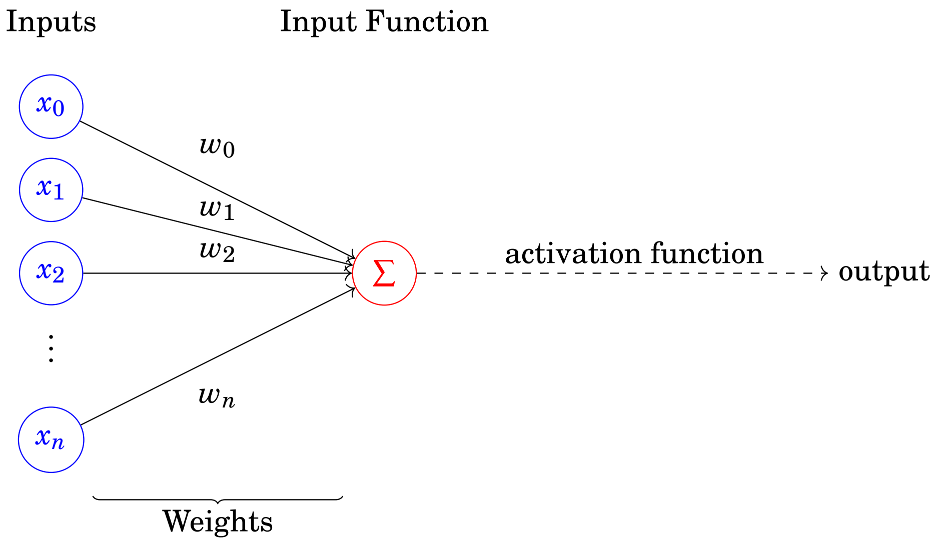


Figure 2.13: The inner workings of a neuron (often called a 'unit' or a 'perceptron')

activation function determines to what extent the signal should progress through the network and affect the output of the network. The activation function can either 'activate' and let the signal pass through, or not activate and let the signal die out.

A network 'layer' is a column of these nodes which act as switches: turning off or on depending on the input and weights. The output of one layer becomes the input to the subsequent layer.

The end goal of a neural network is to either maximise or minimise the *objective function* specified by the user: for example, in the case of image classification, the objective function of a network could be to minimise the error when identifying the presence of a cat in a given image. The weights in the network control which inputs affect the output of the network. These weights are updated each 'epoch' (pass over the training data) using a 'loss function', which calculates the error in a network prediction. By calculating this error during network training, the network can adjust weights in the network accordingly in order to minimise future error.

2.6.4 Model Structures

There are multiple ways of structuring a neural network, and hence many models from which to choose; each excelling in a particular scenario.

2.6.4.1 Perceptron

A perceptron is another word for a 'unit' or 'node' within a neural network. Perceptrons can be used by themselves as a type of linear classifier, that makes its predictions based on a linear predictor function.

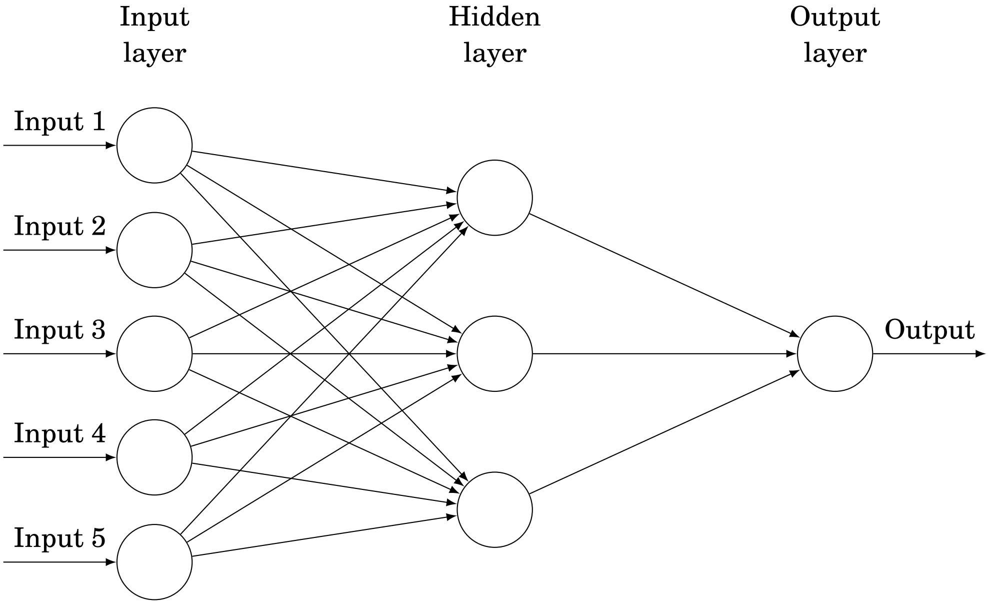


Figure 2.14: An example of a Multi-Layer Perceptron with one hidden layer. Each circle represents a single neuron, whose internal state is depicted in Figure 2.13

More precisely, the perceptron is an algorithm for learning a binary classifier called a *threshold function*: a function that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(\mathbf{x})$ as in Equation 2.16, where $\mathbf{w}$ is a vector of weights, $\mathbf{w}\cdot\mathbf{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$ (where m is the number of inputs to the perceptron), and b is the *bias* (the purpose being to shift the decision boundary away from the origin).

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}\cdot\mathbf{x}+b>0, \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

An illustration of a perceptron is shown in Figure 2.13.

2.6.4.2 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is an example of a ‘feed-forward’ neural network (nodes do not form any cycles). Typically, an MLP has at least three layers: an input layer, a hidden layer (or multiple hidden layers), and an output layer, as shown in Figure 2.14. Each node (bar the input nodes) is a neuron (perceptron) that uses a non-linear activation function (where an activation function of a node defines the output of that node given a set of inputs).

MLP’s use back-propagation for training; this technique is commonly used in networks with multiple hidden layers, and consists of ‘working backwards’ after each epoch to fine tune the

(a) Linearly Separable Data

(b) Non-linearly Separable Data

Figure 2.15: An example of Linearly Separable data and Non-linearly Separable data

weights of the network according to the loss function. The advantage of an MLP over a standard linear perceptron (a machine learning binary classifier) is that it is able to distinguish data that is not linearly separable: linearly separable data is data of multiple classes that can be separated using a straight line, such as in Figure 2.15a. If the data cannot be separated using a straight line, then the data is not linearly separable, such as in Figure 2.15b. Most applications of neural networks deal with data that is not linearly separable, hence the need for more complex algorithms than a standard perceptron.

In practice, smaller MLPs can be very quick to train and test, compared to some of the larger and more complex models. However, the larger the model, the more complex data it can interpret and classify.

2.6.4.3 Convolutional Neural Networks

Convolutional Neural Networks are a powerful form of Multi-Layer Perceptron. They are, in essence, Multi-Layer Perceptrons that use convolution layers and pooling layers, as shown in Figure 2.16. A convolution is a mathematical operation on two functions that produces a third function, which expresses how the shape of one is modified by the other. Convolutional networks are types of neural networks that use convolution instead of general matrix multiplication in at least one of their layers, known as a 'convolutional layer'. Convolution layers take advantage of the local spatial coherence of the input. Because of this, they are widely used in the field of image processing, as they are able to reduce the number of required parameters by sharing weights. 'Deconvolutional' layers effectively apply the inverse of the convolution function. These networks also use layers such as 'Pooling' layers, which use down sampling to reduce the input size, essentially 'compressing' the output of the previous layer whilst conserving 'enough' information. These are often placed following convolutional layers.

In practice, in order to maximise the effectiveness of the convolutional networks, they are required to be very large and complex. As a consequence, they take much longer to train over the simpler MLPs.

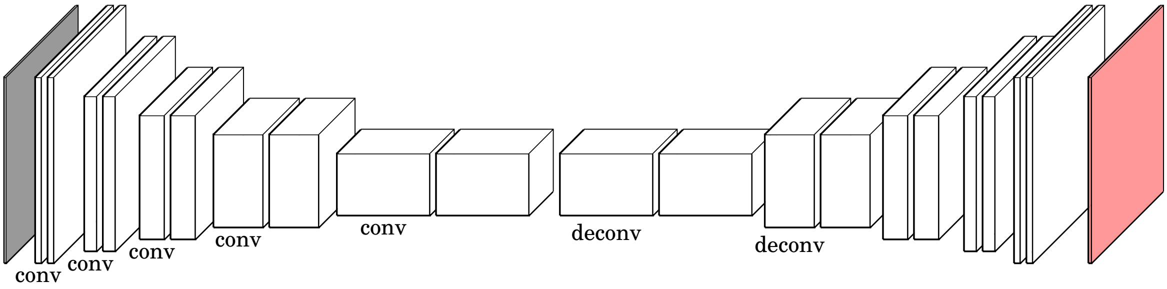


Figure 2.16: An example of a Convolutional Neural Network: each convolutional layer (conv) and deconvolutional layer (deconv) is immediately followed by a pooling layer

2.6.5 Training a Neural Network

We train a neural network similar to an LDA classifier. We provide a **training set**, comprised of two parts:

- **Training Data**, the data to be analysed and classified; for example, the image data for image recognition
- **Training Labels**, the correct value / identity of the training data; for example, the fact that the image is a dog

There are also a number of hyperparameters to consider when training:

- *Model Structure*, to consider before training the network; there is no ‘one size fits all’ model in practice, so finding a suitable model can be a challenging task; in general, models are adapted from related works that aim to solve similar problems, and the precise structure is then tuned manually
  - In case of an MLP, this requires a number of *hidden layers*, each hidden layer containing a number of *hidden neurons / nodes*; again these are adapted from previously trained networks that solve similar problems
- *Learning Rate*, determines how quickly the neural network learns; if the learning rate is too small, the model will converge too slowly; however, if the learning rate is too large, the model will diverge
- *Loss Function*, used to compare the output of the network during training against the intended known output
- *Epochs*, the number of passes over the training data; too few and the model might not train to its full extent; too many and the model may over-fit (lose the ability to generalise in order to minimise the error when classifying the training data)

- *Batch size*, the amount of training data passed to the network in one go; more to do with performance than accuracy of the model
- *Activation function*, the activation function of a node defines the output of that node given a set of inputs, such as ReLU (a linear unit employing a 'rectifier', which takes the positive part of its argument) and Softmax (a function that normalises an input vector into a probability distribution)
- *Optimiser*, used to minimise the objective function; RMSProp [35] is a commonly used optimiser in classification networks

The universal approximation theorem states that "networks with two hidden layers and a suitable activation function can approximate any continuous function on a compact domain to any desired accuracy" [36, 37]. Unfortunately, by only having two hidden layers, one would require an exponentially large number of hidden nodes per layer relative to the input size. Neural Networks are able to solve this by trading off the number of hidden layers for the number of nodes within each hidden layer. Previous work shows that by having fewer nodes per hidden layer, "natural functions" can be learnt quickly [38].

Despite there being a plethora of hyperparameters to consider, literature does not provide a great deal of insight into how these should be selected. The most common approach is to either employ a manual search (in which sets of values are tested at a time, and the trained networks are then compared, whereupon the 'best' networks value is selected), or an automatic parameter selection method is employed (such as grid search or random search [39]).

2.6.6 Validating a Neural Network

The Neural Network is trained on the training data over a number of 'epochs'. Each epoch constitutes a full pass over the training data (in batches specified by the batch size). During this time, the internal weights and biases are constantly being updated to improve the success of the model (with respect to the training data). After each epoch, the network validates itself to check how well the model is learning.

The validation step (or 'evaluation phase') uses unseen data, separate from the training data. This unseen data is supplied to the network, and the network classifies this data, producing an output for each data entry. The output of the network is then compared to the correct labels of the unseen data using the loss function, producing an 'error' metric that informs us how well the network classified the unseen data. This error is often normalised and represented as an 'accuracy' metric: an accuracy of 0 means the model has not learnt anything, and an accuracy of 1 means the network is able to classify the data with 100% certainty.

As the network learns more key features about the training data over a number of epochs, the validation accuracy will increase. There may come a point during training where the validation

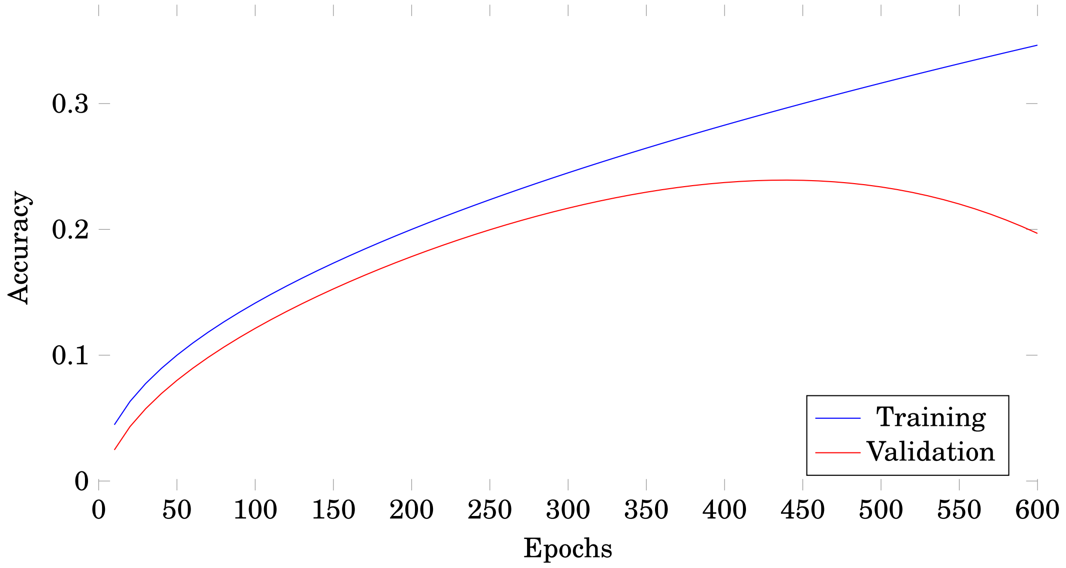


Figure 2.17: Plot taken during training, overfitting occurs after 430 epochs

accuracy reaches a local maximum, and further training epochs start to decrease the validation accuracy. This is known as *overfitting*, where the model loses generality on unseen data by maximising the accuracy on the available training data, as illustrated in Figure 2.17. The training phase should be terminated when the validation accuracy is maximised.

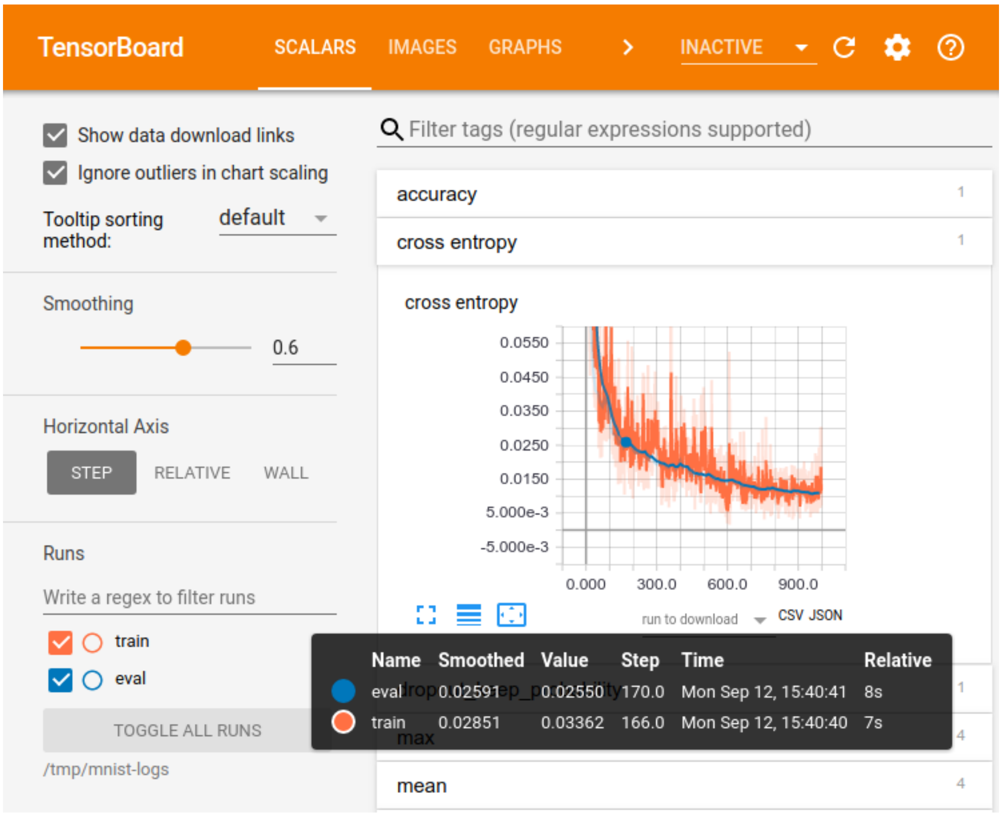
2.6.7 Testing a Neural Network

Once a Neural Network has been fully trained, we can test the network on whatever data we wish. The process is similar to the validation step: we supply the unseen data to the network, and we compare the output to the known labels. The distinction between the validation phase and the testing phase is that the validation phase occurs during training; the testing phase occurs after the network has been trained.

The advantage of testing after fully training a network is that we are not limited to the loss function required in the validation phase: we can use whatever metric we wish to measure the success of our network. Similar to the validation data, it is important that the test data is separate from the training data.

2.6.8 TensorFlow

Google produced TensorFlow [40] in 2015 as a machine learning framework under the Apache 2.0 license. It is available for researchers to use to implement and deploy machine learning models. Although the core of TensorFlow is written in C++, there are front ends available for Python and C++. Previous work in the field of Side Channel Analysis [11, 15] uses TensorFlow as a framework to develop their neural networks.



☑ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

☑ ○ train

☑ ○ eval

/tmp/mnist-logs

	Name	Smoothed	Value	Step	Time	Relative
●	eval	0.02591	0.02550	170.0	Mon Sep 12, 15:40:41	8s
●	train	0.02851	0.03362	166.0	Mon Sep 12, 15:40:40	7s

Figure 2.18: A screenshot showing the TensorBoard user interface

2.6.9 TensorBoard

TensorBoard [41] is a set of tools that allows a visual representation of TensorFlow. It is used to aid in the development and training / testing of TensorFlow programs.

Figure 2.18 shows a screenshot of an example usage of TensorBoard. A network has been trained (or is currently being trained) with a training set, and TensorBoard visually displays various training details in a graphical format:

- The plot shows the training graph, with epochs on the x axis and training loss on the y axis (in accordance to the loss function of the network)
  - The orange line denotes the training loss, and the blue line denotes the validation loss
- In addition to the loss plot, TensorBoard also provides an accuracy plot: a generic metric used to compare models with different loss functions
- There is a slide bar used to ‘smooth’ the plots, allowing for a more visual representation of the training when there are erratic loss spikes during training
- Functionality exists to handle plotting multiple graphs over each other: either overlaying them, or swapping the x axis from epochs to time

- It also includes a graphical representation of the network being trained

TensorBoard is very easy to set up, requiring a single command in the terminal:

```
tensorboard --logdir=./ --host=127.0.0.1
```

The `logdir` argument points to the directory containing the TensorFlow generated log files (this command is run from within this directory), and `host` indicates the IP address on which to host the TensorBoard page.

Chapter 3

Related Work

This chapter describes related work to the field of the thesis. We cover work ranging from the conception of Side Channel Analysis to the more powerful attacks in recent literature. We describe the papers that apply the Belief Propagation Algorithm to Side Channel Analysis in detail. We finally include recent work that utilises Deep Learning in the context of Side Channel Analysis.

3.1 Side Channel Analysis

Kocher et al. Differential Power Analysis [5] As described in Section 2.3, this paper introduced Differential Power Analysis (DPA) to the side channel community. The paper demonstrates power analysis attacks on the DES encryption algorithm. By introducing Simple Power Analysis (Section 2.3.2) and Differential Power Analysis (Section 2.3.3), the authors highlighted vulnerabilities in modern devices, and expressed the need for appropriate countermeasures.

The Kocher et al paper is seminal to the field of Side Channel Analysis, and is cited in almost every power analysis related paper. It also introduced the need for cryptographic devices to be evaluated with respect to side channel attacks, and the need for certification following this evaluation.

Goodwill et al. A Testing Methodology for Side Channel Resistance [42] Due to the threat of effective side channel attacks against commercial devices, manufacturers benefit from providing a guarantee that their product is protected against malicious exploitation. In other words, they wish to prove that their products (microprocessors, embedded devices, etc) are not vulnerable to any form of Side Channel Analysis.

Goodwill et al. propose the use of the Test Vector Leakage Assessment (TVLA) framework to detect the presence of leakage in a target cryptographic device. The framework detects leakage through a suite of Welch's t-tests [43] which partition traces taken from the target device and compare the mean differences in each partition. One example is the 'fixed-vs-random' test, in which the traces are partitioned into two sets: traces taken using a fixed input (plaintext), and traces taken using a variety of different input values. By comparing the difference in the means of both these trace sets, one can infer the presence of data-dependent leakage. The TVLA framework is specified by the ISO standard (ISO/IEC 17825:2016 [44]).

Other methods of leakage detection have been proposed in recent literature. Recent findings show that the mutual information between traces and known intermediates can be used to detect the presence of data dependent leakage [45–47]. The correlation between the traces and a known intermediate can also be exploited [48].

The Common Criteria (CC) [49] and EMVCo [50] evaluation approach is to subject the device to a set of chosen attacks, representing 'all' of the most powerful attacks proposed in recent literature. The chosen set of attacks is managed by the JIL Hardware Attacks Subgroup (JHAS) and is not publicly available.

Whitnall et al. highlight the shortcomings of the methods mentioned above in *'A Cautionary Note Regarding the Usage of Leakage Detection Tests in Security Evaluation'* [51]. One of these shortcomings is the ambiguity of what the proposed methods intend to achieve. If a device passes a leakage detection test, intuition suggests that the device is free from leakage; however, this is not necessarily the case. One can only conclude that no leakage was detected using that specific approach.

The debate of the 'best' method of leakage assessment is still ongoing. Whitnall et al. show that finding the 'best' approach for all evaluation goals is not viable, as each scenario is unique. In this thesis, we consider the 'worst case' adversary: one that is able to exploit all possible leakage points within a set of traces. We believe this is the best way to evaluate our device. In the rest of this chapter we explore the various attacks that have been proposed in recent literature.

The intuition that subjecting the cryptographic attack to the 'worst case adversary' is echoed in work such as *'Soft Analytical Side-Channel Attacks'* by Veyrat-Charvillon et al. [7] (which will be discussed further in Section 3.2), and *'Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database'* by Prouff et al. [15] (which will be discussed further in Section 3.3).

Mather et al. Multi-target DPA attacks: Pushing DPA beyond the limits of a desktop computer [6] When DPA was first published by Kocher et al [5], the computations were initially geared towards low cost equipment and technology. Since then, advances in technology have increased the computational capabilities of the typical adversary, allowing much more powerful attacks. A trade off can now be made between the data complexity and the computational complexity of attacks; in other words, the more powerful resources an adversary has, the fewer traces required for a successful attack. Multi-Target Attacks are an example of how to reduce data complexity. Mather et al. do this by attacking multiple targets (multiple leakage points each associated with a separate intermediate) within a single trace simultaneously, through extensive use of parallelism. The result is an improvement on the classical DPA attack that can harness the power of parallelisation in CPUs and GPUs.

The question of which additional targets (further to the outputs of the `SubBytes` step) to exploit within the algorithm is addressed early in the paper in the context of AES: the output of `MixColumns`, along with the intermediate computations during `MixColumns`. The paper does not extend past the first round of AES. However, if the adversary has access to a large amount of compute power, and wishes to maximise the information extracted from each trace, it poses the question: are the later rounds of AES vulnerable to side channel exploitation? If so, how do we combine that information together with leakage taken from the first round? This question is answered in *'Soft Analytical Side-Channel Attacks'* [7] by Veyrat-Charvillon et al. (which will be described in detail in Section 3.2), and extended further in my work.

Chari et al. Template Attacks [52] Chari et al. propose a variant of DPA in which the adversary has access to an additional tool: a 'replica' of the target cryptographic device. The assumption is that the replica functions identically to the target device, but the adversary has full control over the inputs (including the secret key), and can produce their own traces at will. Using this replica, the adversary can profile the leakage of the device, creating *templates* that characterise the leakage of certain input values. Further details are described in Section 2.3.5. Chari et al. show that Template Attacks are exceptionally powerful, and achieve an improved success rate over standard DPA as introduced by Kocher et al [5].

The power of Template Attacks proposed by Chari et al. could now be applied to the Multi-Target Attack proposed by Mather et al. This would allow the adversary to have a replica of the target device from which they could characterise the leakage of *multiple* leakage intermediates within the cryptographic algorithm. In Section 3.2, we explore papers that do just that. However, in order to combine the information from multiple targets, they use the message passing algorithm known as *Belief Propagation*.

3.2 Belief Propagation in Side Channel Analysis

Veyrat-Charvillon et al. Soft Analytical Side-Channel Attacks [7] This paper introduces Soft Analytical Side Channel Attacks (SASCA) as an alternative way of modelling Differential Power Analysis. The idea is split up into three main steps:

1. Model the target cryptographic algorithm (e.g. AES) as a *factor graph* as described in Section 2.5.2

2. Supply leakage information taken from the target device into the newly constructed factor graph as described in Section 2.5.5

3. Run the Belief Propagation algorithm over the graph, computing the marginal distributions of the key bytes after termination, as described in Section 2.5.3

This represents the (theoretically) *best attack* against a cryptographic device: the idea is that *all possible* leakage points are exploited within the trace, and are combined to extract information on the secret key. However, there have been some gaps in the explanation of the algorithm that make implementation from the description ambiguous. These have been noted as follows:

- In the SASCA adaptation of the belief propagation algorithm, Veyrat-Charvillon et al. note that *"any value that does not leak (either because it is protected*[1] *or precomputed) has a uniform prior [distribution]"*. However, the first round of AES starts with the `AddRoundKey` operation, where the key bytes are XOR'd with the plaintext bytes. If, for whatever reason, at least one of the plaintext bytes does not leak any information, the SASCA algorithm *always fails* (never finds the first-order key). This is due to the fact that the only factor nodes connecting to the key bytes are `XOR` nodes, which also connect to the plaintext bytes. If the plaintext nodes have a uniform probability distribution, *no information* can be passed back to the key bytes. This scenario is only likely in an Unknown Plaintext Attack (where the message input to AES is not known by the attacker). This scenario is not addressed by the paper.

- The paper suggests using a 'backbone' to connect the shared key bytes to each trace used in the attack (such that all shared key bytes are connected to nodes in every trace, creating one huge graph). However, this increases not only the computation complexity, but also the memory complexity; the size of the graph scales linearly with the number of traces, and this large graph must be stored in memory whilst the Belief Propagation Algorithm is performed. This computation would require a large amount of memory to be performed efficiently; it would not be practical on current generation desktop workstations.

- Three functions are given as examples for the factor nodes: XOR, SBOX, and XTIME. Firstly to note, factor functions are *implementation specific*; other implementations of AES might use more complex functions, and as SASCA needs the exact implementation specific factor graph in order to function, these functions would need to be hardcoded into the SASCA algorithm. This is significantly less portable than a standard Template Attack. Secondly, with functions like XOR, the direction of message propagation is not important; XOR nodes have three neighbours, and to communicate a message to one neighbour, one must take the product of the messages coming from the other two neighbours. However,

[1] Protected through the use of a Masking Scheme, see Section 2.3.6.1

the SBOX and XTIME functions are both directed functions (non-involutory), and need extra information (the direction) in order to perform the correct operation on the input (e.g. SBOX(t) = s, but SBOX(s) $\neq t$). These directional requirements are not discussed within the SASCA paper, but must be considered for the implementation of the Belief Propagation algorithm.

- The results reported in the SASCA paper show a 100% success rate when the SNR is 2^4. As attackers, we are interested in recovering the key bytes k_i (Figure 2.7 for reference). Noise affecting the key bytes directly would affect the results considerably more than noise affecting a node far away from the key bytes (e.g. p_{17}). If noise did affect a key byte directly (as it might with an SNR of 2^4), it would be very difficult to recover the key. It is unclear how the 100% success rate the paper achieved is possible in these circumstances.
- Veyrat-Charvillon et al. observed SNRs ranging from 2^2 to 2^{-5}. We note that this SNR range is significantly higher (less noisy) than that observed in practical situations, where we observed SNRs between 2^{-5} and 2^{-6}.

Grosso and Standaert. ASCA, SASCA and DPA with Enumeration: Which One Beats the Other and When? [53] Further to the original SASCA paper, a follow-up paper was published that compared the accuracy of SASCA to ASCA (*Algebraic Side Channel Attacks* [54], developed by the same authors as SASCA) and to enumerated DPA attacks; specifically, a profiled Template Attack.

However, as Grosso and Standaert make clear in the first section of their paper, SASCA is better than ASCA in mostly every aspect, theoretically and practically. ASCA represents the cipher as an instance of an algebraic problem, such as Boolean satisfiability (the problem of determining if there exists an interpretation that satisfies a given Boolean formula). A ‘solver’ is an algorithm or program that aims to satisfy a given problem, and requires equations between variables as input. A single AES trace would be represented as approximately 18,000 equations in 10,000 variables. This already has very large memory complexity. It then feeds in the leakage information to these solvers; however, these solvers require ‘hard information’, that is, no errors. In a practical sense, getting actual data in a noise free setting is next to impossible, and Grosso and Standaert show in the paper how the performance of ASCA drops when attacking traces with noise.

Grosso and Standaert comment that ‘SASCA are in general preferable to ASCA (i.e. both for noise-free and noisy scenarios’ (Section 1 Summary in [53]), as it has a much lower memory complexity by modelling the cipher as a factor graph (similar to a Hidden Markov Model [55]) rather than a series of equations, and it can deal with a significant amount of noise.

However, SASCA has a much larger computational complexity, as creating the factor graph and then performing the Belief Propagation algorithm (especially with multiple traces) is computationally expensive. For this reason, in Grosso and Standaert’s comparison of SASCA with a

Divide and Conquer attack (an attack that first attacks independent parts of the key separately, then combines these pieces of information), the DC attack is allowed to use key enumeration (able to 'check' a large number of keys according to a ranking scheme, allowing success to be more lenient than a first-order attack). More accurately, the success rate of the SASCA attack is compared to Divide and Conquer attacks that exploit *"a computational power corresponding to up to* 2^{30} *encryptions"*.

Grosso and Standaert compare SASCA and ASCA against a Bivariate Template Attack (univariate targets a single leaking operation, whereas bivariate targets two), attacking the `AddRoundKey` and `SubBytes` operations (i.e. $\{s_i\}_{i=1}^{16}$ and $\{t_i\}_{i=1}^{16}$, Figure 2.7 provided as reference). The paper is guaranteed to have a level of bias in its findings, so it is as expected that SASCA *"remains the most powerful attack when the adversary has enough knowledge of the implementation"*. The power of SASCA comes from the number of exploitable leakage samples available to the adversary; the more available, the greater the power SASCA has over Divide and Conquer style attacks.

3.3 Deep Learning

Song et al. Overview of Side Channel Cipher Analysis Based on Deep Learning [56] This short paper (published June 2019) provides an overview on the current state of research regarding Deep Learning applied to Side Channel Analysis, specifically looking at which network models have been utilised in different scenarios. This ranges from the simpler Multi-Layer Perceptrons (MLPs) to the more complex Convolutional Neural Networks (CNNs). This work also provides insight into the current research situation, highlighting the success of published attacks against classical power analysis methods.

Prouff et al. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database [15] Deep Learning has been applied to Side Channel analysis in recent literature. Training neural networks to classify power leakage has many advantages (as described in Section 2.6.2); one that is highlighted in this paper is that CNNs (and also MLPs) are resilient against clock jitter. The networks are successfully able to 'pick out' the useful leakage within a window of power consumption values, and successfully classify a target leakage point. Prouff et al. conclude, after comparing a trained MLP to a trained CNN, that the MLP outperforms the CNN in a jitter-free scenario, whereas the CNN is more resilient against clock jitter. This is expected given the shift invariance of the convolutional operation.

This paper not only provides an in depth analysis on how Prouff et al. trained a neural network to aid in a side channel attack, but it also provides benchmarks upon which other researchers can test their own neural networks. The data used by this paper have been released to the public domain and have been used in the current research as reported in Chapter 6.

Wong et al. Understanding data augmentation for classification: when to warp? [9] Training a neural network is difficult when the training set is small. This could happen in side channel analysis, for instance, if the adversary is unable to acquire a large number of traces in order to build an accurate profile of the target leakage due to restrictions on the physical target device. In this case, one solution is to use Data Augmentation. The idea behind this is to increase the size of the training set by synthetically creating new data from the existing training set.

The paper highlights two approaches in particular: *data warping*, where transformation on the existing training set take place in the data-space, and *over-sampling*, where additional samples are created within feature-space. The paper concludes that data warping yields better network performance than over-sampling.

Cagli et al. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures [10] Cagli et al. showed that the performance of a CNN can be improved when targeting jittery traces by using data augmentation on available training data through a combination of two methods:

1. by 'shifting' a window within existing traces, and
2. by 'add-remove', in which R time samples are selected at random and either 'added' (duplicated) or removed from the trace

By doing this on the training data supplied to the neural network, the hypothesis is that the network will be able to learn how to deal with jittery traces. Their results show that this application technique outperforms an implementation of a Gaussian Template Attack with trace realignment.

Kim et al. Make Some Noise: Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis [11] In this paper, Kim et al. aim to find which CNN structures work best in certain scenarios. To do this, Kim et al. focus on four data sets:

1. The DPA contest v4, a masked AES implementation freely available for public download [57]
2. an unprotected AES-128 implementation
3. an AES implementation with a random delay countermeasure implemented in the software
4. the ASCAD dataset, as defined in [15]

By considering each data set independently, Kim et al. build a CNN that works optimally. The paper contains results that show that the success of the network structures is data dependent (different structures work well on different data sets).

The paper concludes by highlighting the need of a suite of CNN networks, so that people can pick and choose the best structure for their use case. In addition, Kim et al. mention that when Gaussian noise is added to the training data, the network becomes more resilient to noise. The paper includes a demonstration of this using the ASCAD data set as an example.

Martinasek et al. Optimization of Power Analysis Using Neural Network [12] Martinasek et al. focus on the preprocessing side of the Deep Learning-assisted Side Channel Attack; how can one manipulate the training data to develop the best neural network for a given target device? The method proposed in this paper involves finding the average trace within the training data set, and then representing each trace as the difference between itself and the average. The key idea behind this is that the Neural Networks no longer learn arbitrary leakage values, but the distance of the measurements relative to the average, which may be easier for the Neural Networks to learn. Kim et al. demonstrate this technique using an MLP, and show an increase in the first order success rate of a classification attack.

However, this technique does come with some downsides: it *'suppresses alternative probabilities'*. In other words, Martinasek et al. no longer produce a ranking of all possible keys; instead producing a single 'best guess', where all other guesses are equally likely. The effect of this is that the approach does not support backtracking to try other key guesses if the most likely key guess is incorrect.

Martinasek et al. Profiling Power Analysis Attack Based on Multi-layer Perceptron Network [13] Martinasek et al. use the MLP previously developed to compare the success of the network to a Template Attack. To gauge the success rates between the two techniques, Martinasek et al. used the guessing entropy metric. The network structure itself (all hyperparameters chosen) was not included within the paper, and the paper concludes that the Template Attack outperforms the neural network when the data has not been preprocessed. By preprocessing the data (as described in [12]), Martinasek et al. are able to match the success rate of the MLP to the Template Attack.

Martinasek et al. Profiling power analysis attack based on MLP in DPA contest V4.2 [14] Almost a continuation of [12], Martinasek et al. perform the same techniques, but targeting the DPA contest v4.2 trace set (a publicly available trace set used to compare Side Channel Attacks on equal footing). Martinasek et al. is successful in recovering the key from some but not all of the data sets; Martinasek et al. note although unsure why some datasets failed, it was most likely due to a degree of 'distortion' within the dataset.

3.4 Summary of Related Work

The Kocher et al. paper [5] demonstrated the vulnerability of commercial devices by exploiting side channels. Thus, devices must be evaluated with respect to this side channel vulnerability. Idealistically, we wish for a tool that can evaluate a cryptographic device and provide certification detailing its resilience against Side Channel Analysis.

The debate of how best to do this is ongoing. The TVLA [42] + ISO [44] method proposes the use of statistical hypothesis testing as the sole required measure. The CC [49] and EMVCo [50] evaluations involve subjecting the device against a suite of the 'best' side channel attacks proposed in recent literature (although this suite is not publicly available). The papers included in this Related Work chapter propose novel attacks that exploit side channel vulnerabilities using an assortment of methods, including statistical analysis in Section 3.1, inference-based attacks in Section 3.2, and Machine Learning-based attacks in Section 3.3.

In this thesis, we consider the best leakage assessment technique to be subjecting the target device to the 'worst case' adversary, by mounting the theoretically best attack against the device. The resulting attack is a combination of two techniques: the Belief Propagation Attack proposed by Veyrat-Charvillon et al. [7], with the classification phase utilising Deep Learning as demonstrated by Prouff et al. [15]. We improve the feasibility and efficiency of such an attack through detailed experimentation, and compare our improved attack to other attacks proposed in recent literature.

CHAPTER 4

THE BELIEF PROPAGATION ATTACK

This chapter is concerned with the Belief Propagation algorithm, and its application to Side Channel Analysis. It is based on an initial idea proposed by Elisabeth Oswald and notation designed by Arnab Roy. A subset of the content included in this chapter was published at CARDIS 2018, with the title *A Systematic Study of the Impact of Graphical Models on Inference-Based Attacks on AES* [2].

4.1 Introduction

In Section 2.5 we describe how the Belief Propagation (BP) algorithm has been used in Side Channel Analysis. Section 3.2 details the shortcoming of current Belief Propagation Attack (BPA) implementations, such as Veyrat-Charvillon et al. [7]. The limitation of the attack is the large memory requirement, and as such, is not practical for real world scenarios.

In this section we describe our contributions that develop BPA into a more efficient and practical attack. We begin by describing in detail the system built to experiment with BPA written in Python. Our first improvement makes the Belief Propagation algorithm itself more efficient by reducing the practical running time through ‘Epsilon Exhaustion’: a new approach that incorporates a method to identify when information has been adequately propagated throughout the factor graph, thereby enabling an ‘early-out’ to obviate the need for significant further computation that does not add materially to the analysis. Our second improvement detects erroneous traces through the ‘Ground Truth Check’ (by comparing beliefs on known variables). We then show the observations taken by experimenting with this system, starting with a method to connect multiple traces together without incurring a large memory overhead. By connecting multiple traces together, we combine leakage information from multiple traces, significantly reducing the possible key space and increasing the chance of first order success (successful key recovery). We then show that it can be beneficial to remove certain nodes from the graph (to enable a potentially significant reduction in algorithmic and data complexity), and we propose a method to safely remove a node. Finally, we give details on the odd phenomenon of Loopy Belief Propagation (Belief Propagation on a factor graph containing cycles, which prevents convergence): the chaotic fluctuating of information. We go on to show the scenarios in which this phenomenon is observed, why we should prevent this from happening, and a method to prevent it from occurring.

4.2 Implementation of BPA in Python

In order to experiment with the nature of the Belief Propagation algorithm, we elected to implement it in Python 2.7 [58]. We chose Python for a number of reasons: it is interactive, portable, and supports rapid development. For some of the more resource-intensive functions and procedures, we used a Cython [59] compiler, converting the Python into C code before compilation. Compiled Cython functions run faster than functions using the standard Python syntax.

4.2.1 Project Layout

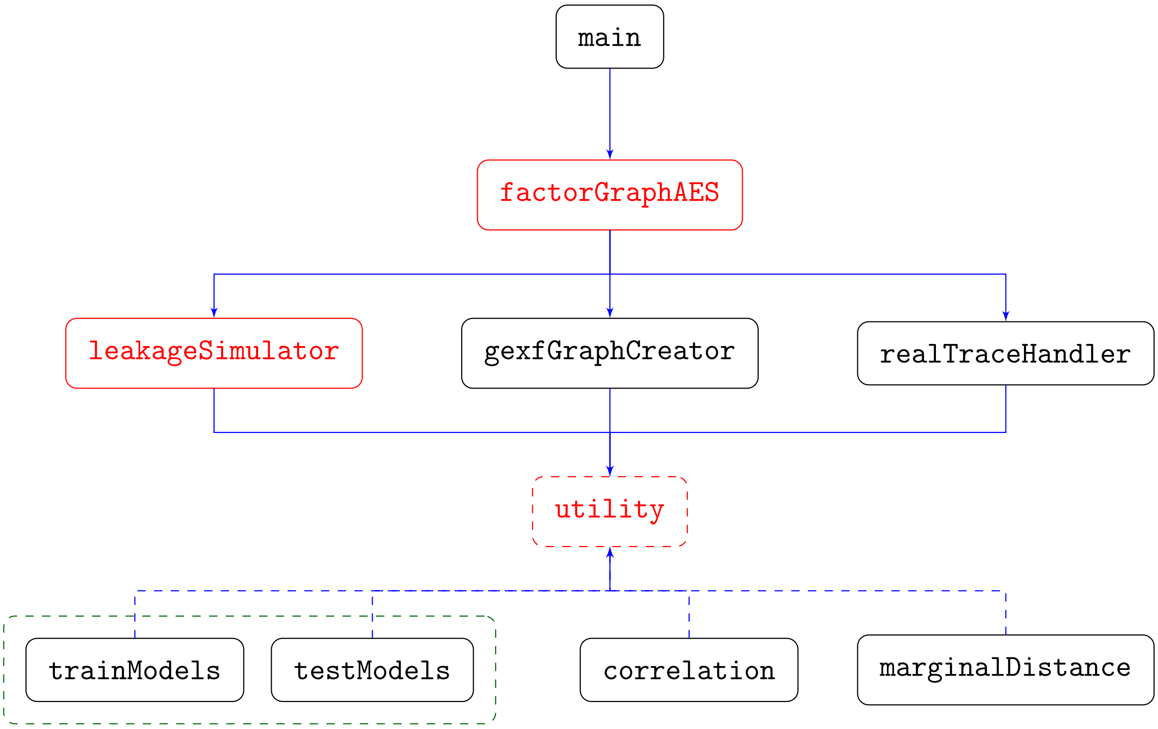


Figure 4.1: `belief_propagation_attack` Project Layout

Figure 4.1 shows the layout of the `belief_propagation_attack` project. Each block represents a Python class file. The blocks in red have a high computational complexity, and are therefore written and compiled using Cython for runtime efficiency (using the suffix `.pyx`). The other blocks are written in Python (using the suffix `.py`). The two classes enclosed in the green box (`trainModels` and `testModels`) are for training and testing the Neural Networks.

4.2.2 `main.py`

The main file, used to run the Belief Propagation Attack. It first handles the argument flags provided on the command line. It then sets up the Factor Graph and Leakage using `factorGraphAES.pyx`, and starts running the Belief Propagation algorithm. The final state of the graph is recorded and stored in corresponding results files as indicated through the argument flags. Without any flags, running `main.py` will run a simple BP attack on simulated data. Table A.1 is provided in the appendix as an exhaustive list of available argument flags sorted into categories.

4.2.3 `factorGraphAES.pyx`

This class file first handles the set up of the Factor Graph by using `gexfGraphCreator.py`. Once created, it then handles the leakage information (described in Section 2.5.5) by either using `leakageSimulator.pyx` if simulating the data using ELMO, or `realTraceHandler.py` if using the leakage taken from a real device. It then handles all Belief Propagation related operations, such as:

- Setting the initial distributions of all leaking variable nodes according to Equation 2.10, and supplying non-leaking variable nodes with a uniform distribution
- Passing messages from variable nodes to factor nodes (and vice versa) according to Equations 2.8 and 2.9
- Storing all distributions as the 'edges' between nodes
- Computing the marginal distributions of any given node, mostly used on the key bytes to determine the rank of the target key
- Implementing the Epsilon Exhaustion termination criterion and Ground Truth Checks, which will be described in Sections 4.4 and 4.5 respectively
- Altering the leakage on certain nodes, removing nodes, and 'fixing' certain variable nodes in a given trace, in order to measure the 'importance' of a node (all discussed in Section 4.7)

We implement the messages that propagate around the factor graph as `numpy` arrays of type `float` and of size 256. `numpy` (mathematical library for Python) supports fast and efficient array manipulation, which is crucial for our implementation of the Belief Propagation algorithm to run smoothly. Each array of floats represents a probability distribution of a specific variable node, and as such, is constantly normalised to sum to 1. The value stored in each array element indicates the probability the variable node has of being the index of the element (as nodes are bytes, the possible values of the index range from 0 to 255).

`factorGraphAES.pyx` can be used independently if one wishes to experiment with the Belief Propagation algorithm without the restrictions `main.py` might enforce.

4.2.4 `leakageSimulator.pyx`

This class file is able to simulate leakage for a specific value, as well as adding Gaussian noise if necessary. This is hard-coded for `AES FURIOUS`, along with a file that can simulate `ARM AES`. The idea behind this modularisation is to allow the use of any generic block cipher.

As discussed in Section 2.4.3, the ELMO tool has the ability to generate simulated power traces from provided source code. In our work, however, we tweak the output of each leaked power value, to match the following equation:

$$\boldsymbol{y} = \delta + [\ \boldsymbol{O_2}\ ]\beta + \epsilon \tag{4.1}$$

Unlike Equation 2.4, we are only concerned with the Hamming Weight leakage on the second operand. Using this simplified power model, we generate our traces using an AES implementation written in the ARM assembly language.

4.2.5 `gexfGraphCreator.py`

Creates the structure of the `AES FURIOUS` graph, using the third party package `networkx` to set up the nodes and the edges. It stores the resulting graph in the `Graphs/` folder, which is then checked (and loaded if present) by `factorGraphAES.py` each time a new graph is required, preventing repeated creation of the same graph in each repetition of the attack.

We implemented one small change to the structure of the graph to how it was originally introduced in the SASCA paper [7]. The leakage information is modelled to be identical to Figure 4.2 (leakage provided to every variable as a leaf factor node), but in our implementation we do not represent the leakage as separate nodes, using 'initial distributions' within the variable nodes themselves. We do this because in the original proposal of the algorithm, these 'Leakage Factor Nodes' are leaf nodes. They constantly send out their message at each iteration, and accept incoming messages from their neighbouring variable node. This is redundant, as their message only needs to be sent out once, and as we never compute the marginals of these factor nodes, we do not need to accept incoming messages. As a consequence, we removed them, replacing them with the initial distributions contained within each variable node. This reduces the number of messages to be sent per BP iteration by the number of variable nodes in the graph (in the case of the full 10 rounds of AES-128, this saves 1,212 operations per iteration).

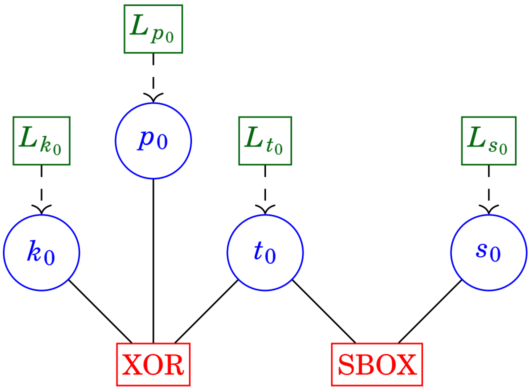


Figure 4.2: Factor Graph example using Leakage Information L_v as Factor Nodes; in this work the Leakage Factor Nodes are removed and instead we use 'initial distributions' stored internally in the variable nodes (in blue)

4.2.6 `realTraceHandler.py`

This is the class used to handle real trace data. It holds a pointer to the trace data in memory (it does not load it due to the trace data being large in size). Additionally, it is able to handle leakage on request, using any specified classifier listed below:

- Gaussian univariate templating, using the templates found in `musigma.dict` (Python dictionary of templates built using `correlation.py`, which will be discussed in Section 4.2.11)
- Linear Discriminate Analysis classifier, using the LDA files located in the `lda` folder
- Neural Network-assisted classifier, using the model files located in the `models` folder

In addition, the class file can also test the effectiveness of a requested classifier / neural network model. This feature is used in the `testModels.py` script.

4.2.7 `utility.pyx`

This Cython file contains utility functions that are used by all of the files in the project. These range from printing and return statistics on various arrays, to mathematical functions that are the heart of the Belief Propagation algorithm. During development of the system, many functions became a bottleneck of computation (for example, the `arrayXOR` function used in Belief Propagation), so in order to speed it up, it was compiled into C using Cython.

For example, the `XOR` operation present in the AES algorithm calculates the xor of two values. In BPA we use the same factor graph that represents the execution of the AES algorithm: the function nodes that correspond to the `XOR` operations perform a function on two probability distributions that replicates the effect of the `XOR` operation.

Listing 4.1: Python code to perform the XOR of two probability distributions

```
@cython.boundscheck(False)
@cython.wraparound(False)
# Function to xor two probability distributions
def fast_xor(np.ndarray [DTYPE_t, ndim=1] v1, np.ndarray [DTYPE_t, ndim=1] v2):
  # Define Cython local variables
  cdef np.ndarray [DTYPE_t, ndim=1] v_xor = get_zeros_array()
  cdef int i, j
  # Index 0 is easy enough, as i^i == 0
  for val in zip(v1,v2):
    v_xor[0] += val[0] * val[1]
  # Loop through other values, multiplying probabilities
  for i in range(256):
    for j in range(i+1, 256):
      v_xor[i^j] += (v1[i] * v2[j]) + (v1[j] * v2[i])
  return v_xor
```

Listing 4.1 contains the function used to calculate the `XOR` operation over two probability distributions, `v1` and `v2`. This function includes a nested `for` loop, iterating through all possible xor values and updating the probabilities accordingly[1]. Due to the nature of the nested `for` loop, this function is the bottleneck of the Belief Propagation algorithm, and needs to be performed a large number of times per BP iteration (equal to the total number of neighbours of all `XOR` nodes in the factor graph, which for the full AES graph is 2148). A great deal of experimentation went into optimising this function, and the best solution was to compile it using Cython, giving a speedup of roughly 300%. Unlike the Python files in the project, all Cython files must be compiled whenever they are modified.

4.2.8 `trainModels.py`

This file was adapted from the ASCAD file of the same name, used by Benadjila et al. to train their models in *'Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database'* [15]. By using argument flags to tweak the hyperparameters of the model to train, we use keras[2] to train a neural network using available trace data. There is currently support for the following models:

- The MLP pretrained on the ASCAD Database

[1] This produces a non-normalised 'likelihood array', which is then normalised by the separate function `normalise_array()`

[2] A Neural Network API, see Section 6.2.1

- The CNN pretrained on the ASCAD Database
- The MLP and CNN using the best model shown in the ASCAD paper
- A generic MLP model where the following hyperparameters can be modified:
    - Number of hidden layers
    - Number of nodes per hidden layer
    - The loss function

and support for the following hyperparameters that affect all models:

- Number of input units (window size)
- Number of training traces and validation traces
- Number of epochs to train
- Batch size of training
- Learning rate
- Option to add jitter to traces to train for misalignment
- Option to use multilabel encoding (see Section 6.4.9)
- Option to train against Hamming Weight (as opposed to identity value)

Details on training a neural network are found in Section 2.6.5, and our experiments using this file are described in Section 6.2.2.

4.2.9 `testModels.py`

The class file used to handle the testing for the neural networks. It sets up an instance of `realTraceHandler` and requests various networks to classify test data; the results are stored in a csv file for ease of comparison.

Details on training a neural network are found in Section 2.6.7, and our experiments using this file are described in Section 6.2.4

4.2.10 `marginalDistance.py`

This separate script is solely used to measure the importance of nodes using the Hellinger distance, as described in Section 4.7. Having run the Belief Propagation algorithm whilst certain nodes are ‘fixed’ (externally set a specific initial distribution), this script collects the outputs of these attacks and compares the difference in final key distributions to each other. Results using this file can be found in Section 4.7.3.

4.2.11 `correlation.py`

This is the Python file used to initially parse the trace `.trs` file (traces taken from the target device) and to extract various sections into separate files. More specifically, it performs the following methods sequentially:

1. Sets up the necessary directory structure for the project (detailed representation will be provided in Figure 5.7 in Section 5.3.6)
2. Saves the metadata file by reading the `.trs` header: number of traces, number of samples, and various codings and offsets for parsing the `.trs` file (further details will be provided in Section 5.3.1)
3. Strips the trace data and the plaintexts into separate files
4. Simulates the intermediate variable identity values using the extracted plaintexts, storing these in separate files
5. Performs correlation analysis to detect the points of interest of all nodes in the trace file
6. Generates all 256 univariate templates for each variable node
7. Trains Linear Discriminant Analysis classifiers for each variable node

Once run on a new `.trs` file, the main Belief Propagation attack can be run on the new trace set.

4.2.12 Miscellaneous Files

Other files included in the project provide information to set up the environment, and are listed as follows:

- `REQUIREMENTS.txt`, a list of all Python dependencies required by this project; generated using `pipreqs`, one can use the `pip install -r REQUIREMENTS.txt` command to install them
- `PATH_FILE.txt`, includes a list of hard coded directory paths for loading traces, neural network models, and other large files; for instance, during development, the trace files were around 20GB in size, so these were stored on an external hard drive, and `PATH_FILE.txt` included a path to these traces
- `README.md`, includes all necessary information to set up the project environment
- `Makefile`, used to build and test the project using `make build` and `make test` respectively
- `setup.py`, used to compile the Cython files into C when running `make build`

4.3 Experimentation and Recording Results

As an example, we run the Belief Propagation Attack using the following command:

```
python belief_propagation_attack/main.py -r 20 -t 10 -rep 5 -snrexp -3
```

This runs the attack using 20 iterations of BP, 10 traces, 5 repetitions (each using a different fixed key and different plaintexts), and a simulated SNR of 2^{-3} (s.t. the noise is 8x more than the signal). Upon termination, the program prints the attack results to the console, as shown in Listing 4.2.

Listing 4.2: Output of the BPA program

```
+++++++++ Key Rank Statistics +++++++++
Max:                 12  (~2^3)
Min:                  1  (~2^0)
AriM:               4.6  (~2^2)
GeoM:                 2  (~2^1)
Med:                3.0  (~2^1)
Rng:                 11  (~2^3)
Var:              14.64  (~2^3)

Total Successes:   1  (   20.0%)
Total Attacks:    50
Failures:          0  (    0.0%)
Maxed Iterations: 50  (  100.0%)
Epsilon Exhaust:   0  (    0.0%)
```

After a single Belief Propagation Attack, the final ranking of the key bytes is recorded and appended to a list. After all repetitions of the attack are completed, this list of key ranks is analysed. We are mostly interested in two aspects of the analysis: the logarithm of the median final key rank, and the attack success percentage.

We choose the median rank over the other metrics as it provides a more stable representation of the attack, where a single noisy trace can easily skew the mean result. As the key rank can be anywhere between 1 and 2^{128}, we use the logarithm of the median rank. An attack is deemed successful if the final key rank of the attack is 1: this is also known as 'first order attack success', and means the attacker has successfully recovered the key. The successful attack percentage is therefore the percentage of how many attacks successfully recovered the key.

4.4 Epsilon Exhaustion

4.4.1 Introduction

The Belief Propagation algorithm as described in Section 2.5 is run for a number of iterations. A number of iterations must be chosen to ensure full information propagation. Previous work has not supplied a method of choosing this number, so in the case for the full 10 rounds of AES-128, we use at least 100 BP iterations.

We denote the number of iterations run as the value $t_{\max}$. In each iteration of BP, all messages are updated. After experimentation during the development of the Belief Propagation Attack, we observed three scenarios of the updated messages:

1. The information continued to propagate around the graph up to $t_{\max}$, with the updated messages being drastically different to the original messages.
2. The information continued to propagate around the graph up to $t_{\max}$, but the messages were only being updated a small amount (diminishing returns of updated information)
3. All information had fully propagated after a number of iterations before $t_{\max}$, and further iterations did not alter any messages

The first scenario (drastic message updating) is an example of the 'chaotic' behaviour that exists when the Belief Propagation algorithm is run on a factor graph containing cycles. This behaviour will be addressed in Section 4.8. However, in this section we will focus on the remaining two scenarios: when the Belief Propagation reaches a state of (or close to) equilibrium.

We hypothesise that if we are able to detect this state of equilibrium from within the Belief Propagation algorithm, then we will be able to terminate BPA without detrimentally affecting the attack success. In the case of complete convergence (Scenario 3), this would be intuitive: further rounds of BP do not affect the probabilities. However, in the case of continuous small message updating (Scenario 2), it becomes much more challenging to predict the optimal BP iteration to terminate the algorithm. We therefore propose two metrics: a threshold ε to control how much information being updated is considered 'small', and a constant ε_s to measure the number of consecutive Belief Propagation iterations that must be under the threshold ε in order to terminate the Belief Propagation algorithm.

4.4.2 Experimentation

After every iteration of the Belief Propagation algorithm (consisting of a Variable Pass and a Factor Pass), we observe the messages being sent to each of the key byte nodes. We compare these messages to the messages sent by the previous iteration (excluding the first iteration of BP). If the difference between these messages is smaller than some threshold ε, then we can infer the information has not been updated. To check this difference, we required a distance metric.

After experimenting with multiple distance metrics (Hellinger distance, KL Divergence, and many others), we found the clearest results using the Euclidean distance metric, as defined in Equation 2.14. If we observe this threshold not being met over ε_s iterations of BP (to ensure we have not terminated too early when information is still flowing), then we can conclude that all the information has propagated through the graph, and we can safely terminate the BP algorithm. The Pseudocode for this method is included within Algorithm 4, and is marked by the comment '*Epsilon Exhaustion* check'.

Algorithm 4: The Belief Propagation algorithm complete with the Epsilon Exhaustion termination criterion and the final Ground Truth check. Pseudocode was adapted from my CARDIS publication *'A Systematic Study of the Impact of Graphical Models on Inference-based Attacks on AES'* [2]

```
function BPA ($\mathcal{G}_{\text{aes}}, \varepsilon, \varepsilon_s, \varepsilon_g, t_{max}, k^*, v_p$)
    /* $k^*$ are the variable nodes corresponding to the key                    */
    /* $v_p$ are the variable nodes corresponding to the plaintext              */
    Initialise the messages as i.i.d uniform random variables
    $count = 0$
    foreach $t \in \{1, \ldots, t_{max}\}$ do
        foreach $(v, f) \in \mathcal{G}_{\text{aes}}$ do
            update $q^{(t)}_{v \to f}$ according to Equation (2.8)
        end
        foreach $(v, f) \in \mathcal{G}_{\text{aes}}$ do
            update $r^{(t)}_{f \to v}$ according to Equation (2.9)
        end
        if $(k^*, f) \in \mathcal{G}_{\text{aes}}, \|r^{(t)}_{f \to k^*} - r^{(t-1)}_{f \to k^*}\|_\infty < \varepsilon$ then
            $count = count + 1$
            if $count == \varepsilon_s$ then                      /* Epsilon Exhaustion check */
                break
        else
            $count = 0$
        end
    end
    if $\|r_{f \to i_p} - m_{\mathcal{L}}[v_p]\|_\infty < \varepsilon_g$ then            /* Ground Truth check */
        return 0                      /* $m_{\mathcal{L}}[v_p]$ is the leakage distribution at node $v_p$ */
    else
        return $-1$                                                  /* Discard trace */
```

When using Epsilon Exhaustion (EE), the Belief Propagation can terminate in one of two ways: either reaching t_{max} iterations, or terminating early when the updated information is below threshold ε. We record the ratio of the BP termination method when we modify the values of ε, as well as documenting the attack success with the median final key rank. We set the value of ε_s to be 10, chosen to be not too small (which would be prone to scenarios where information fluctuated steadily before rapidly escalating), and not to be too big (which would not save a significant amount of runtime).

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100
        -rep 100 -raes 1 -snrexp [-1, -7]
-epsilon_s 10 -epsilon [0.1, 0.01, 0.001, 0.0001, 1e-5, 1e-6]
```

4.4.3 Results

Epsilon ε	% Success	Median Rank	% EE Termination	Average Trace Time (s)
None	95	0	0	10.04
0.1	99	0	42.48	7.2
0.01	96	0	30.8	8.87
0.001	96	0	26.11	9.26
0.0001	96	0	22.45	9.33
1.00E-05	96	0	19.13	9.58
1.00E-06	96	0	16.82	9.88

(a) Epsilon Exhaustion Results for SNR = 2^{-1}

Epsilon ε	% Success	Median Rank	% EE Termination	Average Trace Time (s)
None	0	10	0	9.04
0.1	0	10	100	3.39
0.01	0	10	100	4.39
0.001	0	10	100	4.76
0.0001	0	10	100	5.39
1.00E-05	0	10	100	5.67
1.00E-06	0	10	100	7.17

(b) Epsilon Exhaustion Results for SNR = 2^{-7}

Table 4.1: Epsilon Exhaustion Results in a Low Noise scenario and a High Noise scenario

4.4.4 Observations

Table 4.1a shows the attack results when the SNR is high (relatively small amount of noise). The first observation to note is that we increase the attack success by using the Epsilon Exhaustion termination method.

We also observe a faster runtime of the attack when we use the Epsilon Exhaustion termination method. Our fastest times come from a larger epsilon ε, which is intuitive: the larger the threshold of 'similarity', the sooner they will be detected and the sooner BP can be terminated.

We do not see a notable difference in attack success in this low noise scenario when modifying the value of epsilon ε.

Table 4.1b shows a much noisier scenario, in which first order success is not achieved, but the median rank within 2^{10}. The termination rate and the success of the attack are identical among the experimented epsilon values. Similar to the low noise scenario, we observe the fastest runtime when using a large epsilon ε.

4.4.5 Conclusions

The results of both experiments show that by changing the value of ε, we change the runtime of the Belief Propagation algorithm without significantly affecting the final key results. In the low noise case, we *improve* the success of the attack. This is most likely due to the number of cycles in the graph: when Belief Propagation continues to propagate through a cycle, information seems to fluctuate almost 'chaotically'. This will be discussed further in Section 4.8.

Our findings suggest to use the Epsilon Exhaustion method whenever running the Belief Propagation Attack. In our work, we opt to use an epsilon ε of around 0.01. This gives us a large runtime speedup without the risk of losing information from terminating too early.

4.4.6 Benefits of Method

The benefit of the Epsilon Exhaustion method is a faster attack runtime, without any cost to the attack success. Taking the experiments included in this thesis as an example, we perform the Belief Propagation Attack using 100 traces, with 100 iterations of BP per trace, and an SNR of 2^{-7}. We also wish to repeat the experiment 100 times in order to find the attack success percentage.

If we rely on the $t_{\max}$ termination criterion, an attack would take $9.04 \cdot 100 \cdot 100 \approx 25$ hours. Using Epsilon Exhaustion, we reduce the time taken by half, down to $4.39 \cdot 100 \cdot 100 \approx 12$ hours.

4.5 Ground Truth Checking

4.5.1 Introduction

When we take power traces from a physical device, we expect a certain amount of random noise. This means that not all traces will have equal noise; depending on the device, some traces will be noisier than others. In addition to noise, one small clock jitter in a trace can make it very difficult to extract useful information. One open question in Template Attacks is how the adversary differentiates a 'good' trace (one that provides useful information) from a 'bad' trace (one that provides little to no information, or in some cases, erroneous information).

Within Belief Propagation, the marginal distribution of a variable node is defined as the product of all incoming messages, and multiplied by the initial distribution of the variable node, as described in Section 2.5. We perform this computation on the key bytes after the Belief Propagation algorithm terminates, and we combine the marginals of all the initial sixteen key bytes to produce a ranking of the possible keys. In this context, the messages take the form of probability distributions regarding each key byte. It follows that if a message was sent to a key byte where the probability for the correct value is set to 0, then this will persist after the product is taken with the other distributions. This results in a failure for that key byte, as there is no way to increase the probability of the correct key value. In order to mitigate this scenario, we introduce a novel way to detect an erroneous trace (one that will most likely predict the correct key value as 0).

In this work, we assume that the adversary knows the plaintexts used in the attack traces, referred to in the Side Channel Analysis literature as a 'known plaintext attack'. These known plaintext bytes are referred to as 'ground truths', as we know their values with certainty. The messages sent to each plaintext byte will be a combination of information from the rest of the trace, excluding the 'true' information we know about the plaintext byte. Once BP terminates (allowing information from all over the trace to propagate to the plaintext bytes), we compare the 'belief' of the plaintext bytes to our knowledge of the correct values. For a 'good' trace, we expect these two distributions to be similar. If the two distributions are not similar, however, then this implies that erroneous information has been propagating around the factor graph. Although we cannot locate the source of the erroneous leakage, we can prevent the error from detrimentally affecting our attack success by simply discarding the trace. The 'similarity' between two traces is measured using the Euclidean distance metric. This is done in a similar way to the Epsilon Exhaustion method, as described in Section 4.4.

Due to the structure of the `AES` Factor Graph, the original 16 plaintext bytes $p_0, \ldots, p_{15}$ are leaf nodes, each connected to an `XOR` factor node, which in turn connects to a key byte k_i and an `AddRoundKey` output t_i (as can be seen in Figure 2.7). In order for information to propagate through the `XOR` node to the plaintext byte, neither of the messages passed from k_i or t_i can be uniform. If at least one is uniform, we encounter a 'locking effect'; the `XOR` of a probability distribution with a uniform distribution is always another uniform distribution. This prevents any information from propagating through the `XOR` node. In order for the Ground Truth Check to work, we require some initial leakage to be assumed on the sixteen initial key bytes. If we do not have access to the initial leakage, this method would also work if leakage was provided from the inclusion of the key schedule: the key schedule provides an alternate source of information derived from the round keys that connect directly to the sixteen original key bytes.

4.5.2 Experimentation

In order to show the effect and advantage of implementing the Ground Truth Check, we firstly need to demonstrate the effectiveness with which the Ground Truth Check works, and then we need to show the improvement it gives to the success of the attack. To simulate erroneous traces, we selected all the nodes in 50 out of the 100 traces to have ‘errors’: instead of leaking the correct value, they leak completely random values. Our hypothesis is that the Ground Truth Check can detect these with different levels of Gaussian noise and, when detected, can discard these erroneous traces to improve the success of the Belief Propagation attack.

The Pseudocode for this method is included within Algorithm 4, and is marked by the comment ‘*Ground Truth* check’.

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100
     -raes 1 -bl [k,t,s,mc,xt,cm,h] -blt [0,2,4,...,96,98]
               -snrexp [-1,...,-7] [--IGT, <none>]
```

4.5.3 Results

SNR	Erroneous Traces Detected (%)
2^{-1}	99.88
2^{-2}	99.94
2^{-3}	99.86
2^{-4}	94.44
2^{-5}	34.28
2^{-6}	0
2^{-7}	0

Table 4.2: Percentage Detection Rate for Ground Truth Checking with varying SNRs

	Ground Truths Off		Ground Truths On	
SNR	Success (%)	Median Rank	Success (%)	Median Rank
2^{-1}	78	0	86	0
2^{-2}	0	14	99	0
2^{-3}	0	64	31	2
2^{-4}	0	82	0	34
2^{-5}	0	87	0	84.5
2^{-6}	0	88	0	88
2^{-7}	0	82	0	82

Table 4.3: Results of Attacks varying SNR comparing the Ground Truth checking method

4.5.4 Observations

Table 4.2 shows how many erroneous traces were detected with different levels of noise. With low noise, almost all erroneous traces were detected. This becomes increasingly difficult with large amounts of noise, to the point where no errors were detected with an SNR below 2^{-6}.

Table 4.3 shows the improvement of using the Ground Truth Check over leaving the erroneous traces within the Belief Propagation algorithm. For SNR = 2^{-2}, we get first order success when using the Ground Truth Check, improving the 0% success rate achieved when we do not use the ground truth checking method. However, this is only true when errors are actually detected; the success rates are identical for SNRs of 2^{-6} and below.

4.5.5 Conclusions

This experiment shows that the Ground Truth Check does not detrimentally affect the success of the attack. As the time it takes to perform this check is very small, it is advantageous to use the Ground Truth Check whenever performing the Belief Propagation attack with real trace data. To generate the experimental results shown in the remainder of this thesis, the Ground Truth Check is permanently turned on.

4.5.6 Benefits of Method

The Ground Truth Check improves the attack success of BPA by discarding erroneous traces, allowing first order success in low SNRs (2^{-3} and 2^{-4}). Experimental results show this is especially effective in low noise scenarios. As this method is only performed once at the end of BP for each trace, there is a negligible runtime cost in employing the Ground Truth Check.

4.6 Trace Connecting Methods

4.6.1 Introduction

In reality, side channel adversaries usually have access to a number of traces. In most cases, these traces all use the same fixed key, but with different plaintext values. By combining these traces, one can more accurately extract information regarding the secret fixed key. In the context of Belief Propagation, Veyrat-Charvillon et al. [7] discuss a method to connect multiple leakage traces together. However, there are multiple ways of connecting traces, each with their own situational use advantages. This section first analyses the combination method as it was initially proposed, and then explores alternative trace connection methods that yield superior results.

In this section, we use small factor graphs to illustrate the combination techniques. Figure 4.3 represents a single trace, composed of an `XOR` (representing the `AddRoundKey` step of AES) and an `SBOX` (representing the `SubBytes` step of AES). Note that this is an undirected graph, allowing messages to flow in any direction.

4.6.2 Large Factor Graphs (LFG)

The combination method as initially proposed by Veyrat-Charvillon et al. [7] involves having a single large graph. This large graph is the combination of all leakage traces, connected through

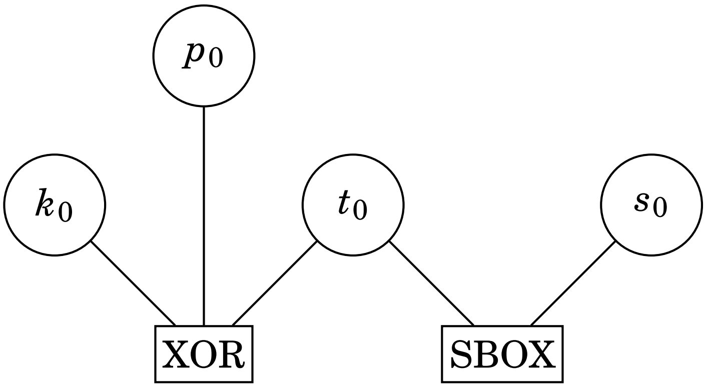


Figure 4.3: An example of a Factor Graph

mutual nodes. In this work we consider cryptographic encryption algorithms that use the same fixed key over multiple encryptions (in our case, AES, but common in most encryption algorithms). The fixed key bytes are therefore common variable nodes across multiple traces, and one can build any number of subgraphs all extending from the initial sixteen key bytes. Figure 4.4 provides a visual representation of this connection method, connecting two traces to the common key byte k_1.

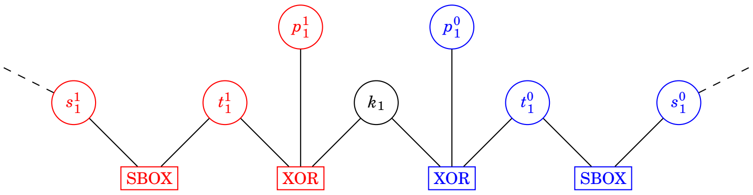


Figure 4.4: Connecting two (or more) traces to form a large factor graph. The blue and red nodes correspond to two different factor graphs (traces) where the node k_1 is common to both of them

By connecting traces together in this way, information from one trace can propagate into another. In the context of Belief Propagation, allowing all available information to propagate throughout the factor graph is known to produce the most accurate results. However, modelling a Large Factor Graph in memory is intensive, and the required memory (and the runtime of the attack) grows linearly with the number of traces. The Ground Truth Check explained in Section 4.5 can also not be applied here, as it would be difficult to pinpoint the source of the error due to inter-trace propagation. Convergence is not guaranteed in this scenario; there will be multiple cycles created by connecting traces together, even if the factor graph for each individual trace is acyclic.

4.6.3 Independent Factor Graphs (IFGs)

Instead of creating one large graph, another option presents itself; perform the Belief Propagation algorithm on each trace in isolation. Only one factor graph would need to be stored in memory at any given time. The resulting set of distributions produced from each independent trace can be combined using Bayes theorem.

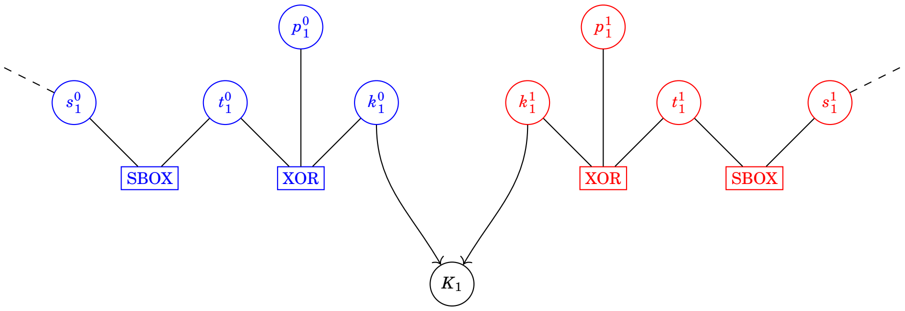


Figure 4.5: Two independent traces connected via an isolated key node. The blue and red nodes correspond to two different factor graphs (traces) where the node K_1 is common to both of them

Constructing the factor graphs in this way allows for straightforward parallelism. In addition, by not connecting the traces together, no additional cycles are created in the graphs, which are known to decrease the likelihood of convergence (Section 2.5). The caveat is that this method prevents the propagation of information from one trace into another, as evidenced by the results of Section 4.6.6. In order to counter this disadvantage, we are able to use the Ground Truth Check (Section 4.5) to discount erroneous traces.

In Figure 4.5, the key nodes k_1^0 and k_1^1 may have different initial leakage on them, for two unique power traces. However, because the true value of this variable is identical among all attack traces (fixed key, varying plaintexts), we are able to average all power values for each of the initial 16 key bytes. By doing this, we average out the noise and improve our attack success.

4.6.4 Sequential Factor Graphs (SFGs)

As mentioned previously, the Large Factor Graph has the advantage of allowing inter-trace propagation; that is, leakage information from one trace has the ability to propagate into another connected trace, which is known to be advantageous in Belief Propagation. The Independent Factor Graph method does not retain this advantage, as it separates the traces from one another. A hybrid of these two methods is the Sequential Factor Graph method, in which the marginal distributions of the key bytes in the $i-1^{\text{th}}$ trace (after termination of the Belief Propagation algorithm) then become the *initial* distributions for the key bytes in the i^{th} trace, as shown in Figure 4.6.

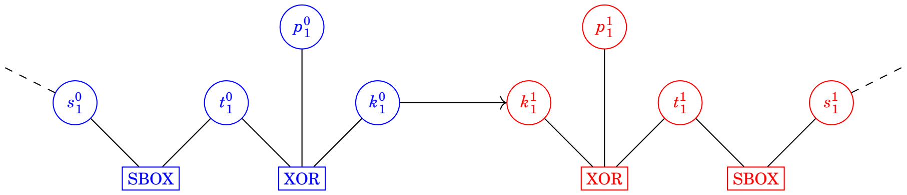


Figure 4.6: Two independent traces connected one way through subsequent traces. The blue and red nodes correspond to two different factor graphs (traces) where there are no common nodes

The advantage of this method is that, similar to the Independent Factor Graph method, it only requires a single trace to be stored in memory during computation, and can easily be made acyclic by removing certain nodes. One can also employ the ‘Ground Truth’ check using the Sequential Factor Graph method. However, unlike the Independent Factor Graph method, the ‘output’ of one trace becomes the ‘input’ to the next, so it is not strictly feasible to parallelise BPA on multiple traces.

4.6.5 Experimentation

The command used to generate these results was as follows, using red to indicate the modified parameters:

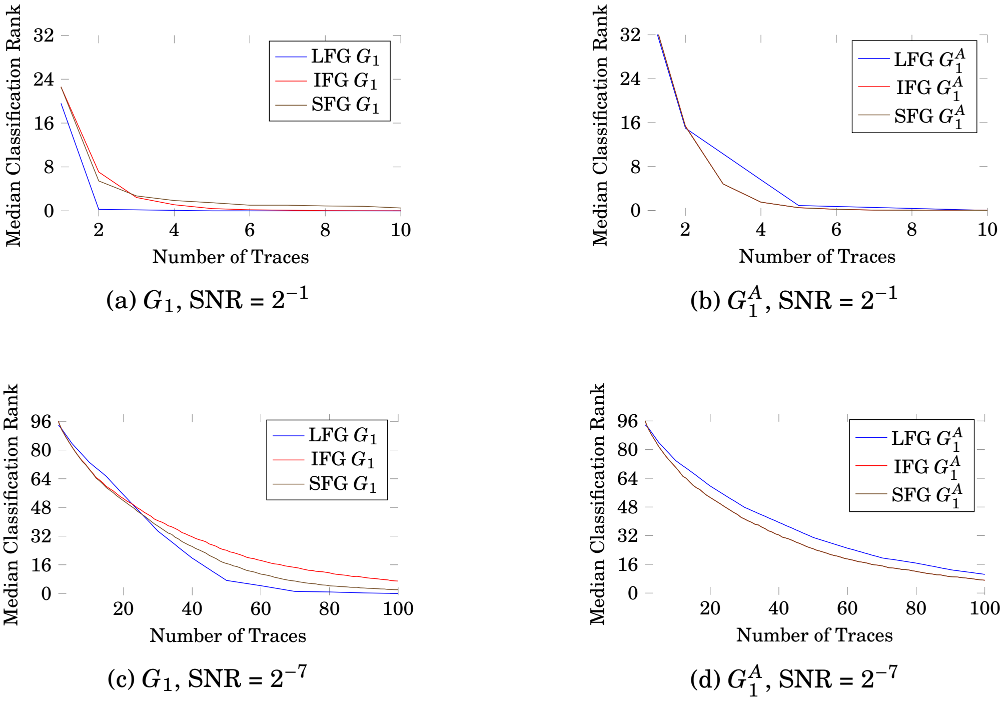
- For the Independent Factor Graphs and the Sequential Factor Graphs:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100 -raes 1 -snrexp
          [-1, -7] [<none>, --RM_C] [<none>, --IFG --SFG]
```

- For the Large Factor Graph method:

```
python belief_propagation_attack/main.py -r 100 -t [1, 2, ..., 100] -rep 100
       -raes 1 -snrexp [-1, -7] [<none>, --RM_C] --IFG --SFG
```

4.6.6 Results



(a) G_1, SNR = 2^{-1}

(b) G_1^A, SNR = 2^{-1}

(c) G_1, SNR = 2^{-7}

(d) G_1^A, SNR = 2^{-7}

Figure 4.7: Comparing methods of graph combination using cyclic graph G_1 and acyclic graph G_1^A, using an SNR of 2^{-1} and 2^{-7}

4.6.7 Observations

In both the low noise (SNR = 2^{-1}) and the high noise (SNR = 2^{-7}) scenarios, the LFG method outperforms the graph connection methods when the graph is cyclic (G_1), but is outperformed by IFG and SFG in the acyclic case. The IFG and SFG methods share identical results apart from in the high noise cyclic scenario, where the SFG method performs slightly better.

4.6.8 Conclusions

The cost of running Large Factor Graphs with a large number of traces may not always be computationally feasible, depending on the attack setup; for example, it costs approximately 13GB memory to use the Large Factor Graph method with 2,000 attack traces (which may be required for protected implementations of AES, see Section 2.3.6). It also prevents the use of the Ground Truth Check, and is therefore unable to prevent the propagation of erroneous leakage. However, when using cyclic graphs, the Large Factor Graph connection method should be considered, as the results show it outperforms the other two combination methods.

When using acyclic graphs (the benefits of removing cycles are suggested in Section 4.8), the experimental results indicate that it is more advantageous to use either the SFG or IFG methods, as these have better success than an acyclic LFG graph. In this case, if the user would opt to use IFG if their computational system could make use of parallelisation (and/or fast runtime is preferable over attack success), otherwise one would choose the SFG method.

4.6.9 Benefits of Methods

To summarise, the following table lists the advantages, disadvantages, and when to use each of the three experimented graph connection methods:

	LFG	SFG	IFG
Pros	Best success rate	Can use Ground Truths	Can use Ground Truths Easy to parallelise
Cons	Large memory overhead Cannot use Ground Truths	Cannot parallelise	None
Use	Small number of traces, small factor graph	Large number of Traces, Speed not important	All-purpose

4.7 Removing Nodes

4.7.1 Introduction

The attack proposed in *'Soft Analytical Side-Channel Attacks'* [7] by Veyrat-Charvillon et al. requires 1,212 variable nodes and 2,756 edges, totalling approximately 6.6MB *per trace* to be held in memory for the duration of the attack. Veyrat-Charvillon et al. experiment with different numbers of traces used during the attack, ranging from 1 to 5,000. The attack that uses 5,000 traces therefore requires 33GB of memory. Depending on the adversary's computational setup, this attack may not be feasible in practice.

In this section we challenge whether it is beneficial to include every node in the full AES factor graph. Our hypothesis is that not all rounds of AES are beneficial to our attack; that is, nodes that are close to the initial sixteen key bytes (e.g. the first round of AES) provide more information than nodes in further rounds (e.g. the second round of AES and beyond). Excerpt taken from page 41, Section 3.5 of *'The Design of Rijndael'* [20] (the creators of Rijndael):

"Two rounds of Rijndael [AES] provide 'full diffusion' in the following sense: every state bit [bit of the current state] depends on all state bits two rounds ago, or a change in one state bit is likely to affect half of the state bits after two rounds."

As described in Section 2.2, the 'state bytes' refer to the 16 bytes that are continuously updated by the operations in AES. This quote describes that the 'full diffusion' property of AES means that if we were to change one bit in the plaintext (given as input to the AES algorithm), this would affect half of the state bits after two rounds of AES. This does not necessarily imply that nodes located after the diffusion state (after two rounds) fail to provide their information through Belief Propagation. We would like to be able to measure the amount of information a single node provides to the Belief Propagation attack. In this way, we will be able to quantify just how 'important' that node is relative to other nodes in the factor graph. An 'important' node is one that when we change its initial distribution, it affects the attack success (by altering the marginal distribution of at least one key byte). We measure the 'importance' of a target variable node by comparing the 'distance' of the target distribution from the distribution of the key bytes, after 'fixing' the value of the target over all possible targets. Each variable node has its own discrete distribution, so the metric we use must be able to work using discrete distributions.

As AES is a deterministic encryption algorithm, the value of an intermediate is fixed relative to the plaintext and the secret key. If, in a given scenario, we know node X_i has value x for a given trace, we can describe this with a probability distribution such that $P(X_i = x) = 1$ and $P(X_i \neq x) = 0$. This method is also used if we need to 'fix' node X_i to have a certain value; we use this in our metric to quantify importance.

Suppose we run the Belief Propagation attack using standard leakage (using an SNR of around 2^1, which only provides a small amount of noise). After termination, we compute the marginal distribution of key byte K as $m(K)$. We then run the Belief Propagation attack twice more, but this time we 'fix' node X_i to have different values: in one attack, we fix node X_i to have value x, which corresponds to the 'correct' value of node X_i. In the other attack, we fix node X_i to have value $x\prime$, which corresponds to an 'incorrect' value of node X_i. When we compare the marginal distributions of key byte K for both of these attacks, we expect the marginal distribution from the attack using the correct value of node X_i to be closer to $m(K)$ than fixing the incorrect value to node X_i. In order to measure the distance between $m(K)$ and $m(K|X_i = x)$, we use *Hellinger distance*.

4.7.2 Importance of a Node

The importance of a variable node X is defined in Equation 4.2, where $D(p,q)$ is the Hellinger distance between the distributions p and q, and $\mathscr{I}$(X) is a set of 'distances' for different values x of X. The Hellinger distance metric is defined in Section 2.5.6.2.

$$\mathscr{I}(X) = \{D(m(K), m_{X=x}(K))\} \tag{4.2}$$

The Hellinger distance (closely related to Bhattacharyya distance) is an appropriate metric in this context as it is used to quantify the similarity between two probability distributions. Our use case is to measure the similarity between two distributions of the same node (a key byte) in two scenarios: one where we do not tamper with the leakage, and one where we 'fix' the leakage of a node to have a certain distribution.

4.7.3 Experimentation

Now that we have a metric that we can use to measure the importance of a variable node (relative to the key bytes), our task is to iterate over all variable nodes in the full AES factor graph and calculate their importance, observing how they differ in importance. To run these experiments, we used our simulated data (in this case, Hamming Weight based leakage) with a small amount of noise (SNR = 2^1).

We ran the Belief Propagation algorithm for 50 full iterations (without breaking early through epsilon exhaustion (see Section 4.4) as this would have affected the results). Following the definition of the metric, we collected the key distributions following termination of BP with the standard leakage, before fixing our target node to have a certain value (repeating the experiment 256 times to simulate all possible values of the node). This allowed us to compute the distance between the non-fixed distributions and the fixed distributions, as shown in Equation 2.15.

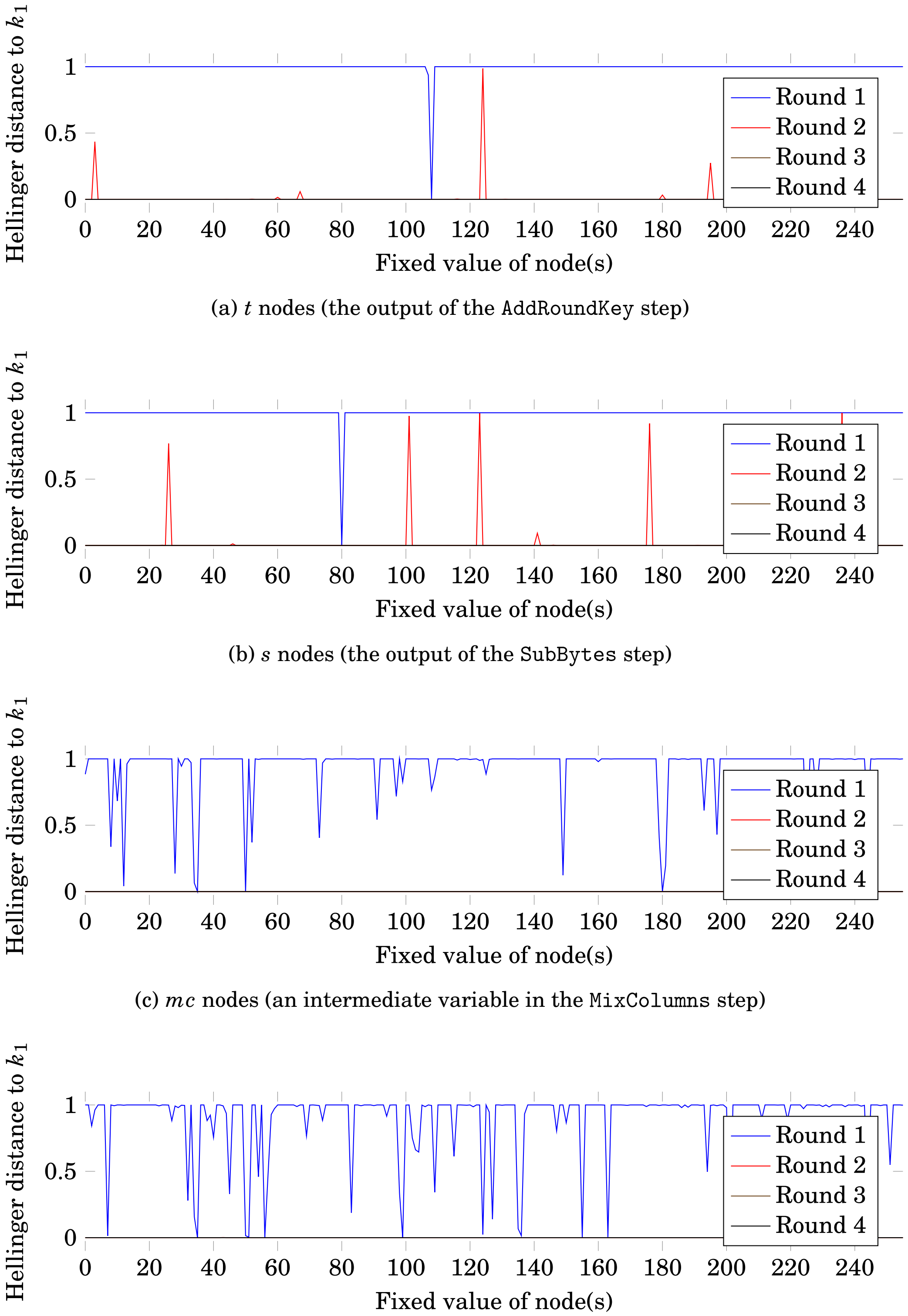
4.7.4 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 50 -raes 10
--TRACE_NPY -snrexp 1 -fix [s001, ..., s032, t001, ...]
```

These experiments produce .npy files which represent the marginal distributions of the key bytes after fixing certain nodes with different values. These .npy files are then analysed using the Hellinger distance metric, using the following code:

```
python belief_propagation_attack/marginalDistance.py --CSV
```



(a) t nodes (the output of the AddRoundKey step)

(b) s nodes (the output of the SubBytes step)

(c) mc nodes (an intermediate variable in the MixColumns step)

(d) cm nodes (an intermediate variable in the MixColumns step)

Figure 4.8: Plots showing the 'importance' of various nodes by computing the Hellinger distances to the key byte k_1

4.7.5 Observations

When we run the Belief Propagation Algorithm with some provided leakage data, after termination of the algorithm we can compute the marginal distribution of the key byte k_1. If we rerun BPA, but this time modifying the initial distribution of some target node, we can compare the 'new' marginal distribution of the key byte k_1 to the one computed when we do not tamper with the leakage. This comparison uses the Hellinger Distance, and produces a value between 0 (the probabilities are identical) to 1 (the probabilities are extremely different). The four plots in Figure 4.8 show this approach for different target variables, over all 255 possible values of the target node.

Figure 4.8a shows the Hellinger distances from the `AddRoundKey` output nodes to the key byte k_1, when fixing the `AddRoundKey` output nodes to different values. The correct value of t_1 (in the first round of AES) in this experiment was 108. This can be seen in the graph, as when t_1 is fixed to this correct value, the Hellinger distance drops to 0 (indicating identical probability to the 'standard leakage'). In all other cases, the node t_1 has so much influence over the key byte k_1 that all other Hellinger distances are 1. This is a clear example of a very important variable node.

If we look at t_{17} (located in the second round of AES), we see that the Hellinger distance is 0 for most fixed values. This is interpreted as follows: no matter what value we fix this node to be, it has very little effect on the key byte k_1 (and all other key bytes). This is an example of a node that is not very important (provides little information, but still some).

Of course, nodes that have a Hellinger distance of 0 for all fixed values (e.g. $t_3$3 in the third round) have no effect on the key byte k_1 at all. This is an example of a node that can be removed from the factor graph without affecting the results.

If we only consider the first round of AES (e.g. t_1, s_1, mc_1, cm_1), we see that the 'importance' of t_1 and s_1 is identical. This is because both t_1 and s_1 are connected to key byte k_1 through a 1-to-1 mapping: Figure 2.7 shows that t_1 is connected to k_1 through an `XOR` node, but as this is connected to the known plaintext byte p_1, this means the message from t_1 is permuted (and not combined with any other information, as would be the case for a standard `XOR` factor node). Similarly, s_1 is connected to t_1 through a 1-to-1 mapping (the `SBOX` factor node).

Figures 4.8c and 4.8d show the Hellinger distances for nodes mc_1 and cm_1 respectively. We see that there are more 'spikes', which indicate occasions when the Hellinger distance was close to 0. As indicated earlier, this shows that they have less influence over key byte k_1, and are therefore less 'important'. This is because they are far away from the key byte k_1, and unlike t_1 and s_1, are connected via at least one `XOR` where the message must be combined with another message before being passed on.

4.7.6 Conclusion

If the marginal key distributions are affected when a target node is fixed to different values, we conclude that the target is 'important'. Nodes that are extremely close to the key distributions (e.g.

the AddRoundKey output t and the SubBytes output s) have a large effect on the key distributions when they are changed; so much so that if these nodes are fixed to any incorrect value, successful recovery of the key is incredibly difficult. We see this with the distance of 1 for all incorrect values, and a distance close to 0 for the correct value. Nodes that are slightly further away (e.g. nodes in the second round) start to lose importance; for many fixed values, the distance between the non-fixed distribution is close to 0, but for a few fixed values we see distances up to 1. Although they have less 'importance' than nodes in the first round, they still maintain the ability to influence the key bytes.

However, nodes in further rounds (e.g. AES round 3 and beyond) have zero effect on the key distributions, regardless of their value. By applying this metric to all nodes in the graph, we can identify nodes with zero importance, and remove them from the graph.

G : the graphical representation of the full ten rounds of AES, excluding the key schedule[3], as defined in Section 2.2. One graph contains 1,212 variable nodes, 1,020 factor nodes, and 2,756 edges, and requires approximately 6.6MB to store the graph of a single trace in memory (if using the Large Factor Graph method, this requirement would scale according to the number of traces required).

G_1 : the graphical representation of the first round of AES. This representation does not include the key schedule. One graph contains 140 variable nodes, 108 factor nodes, and 292 edges, and requires approximately 0.7MB to store the graph in memory. A visual representation of this graph is shown in Figure 4.9, limited to the first column due to space restrictions. Several factor nodes are drawn in red in this graph. Removing them leads to G_1^A.

G_2 : the graphical representation of the first two rounds of AES, up to the SubBytes output of the second round. One graph contains 188 variable nodes, 140 factor nodes, and 372 edges, and requires approximately 0.9MB to store the graph in memory.

G_1^A : the graphical representation of the first round of AES with certain nodes removed (shown in red in Figure 4.9), resulting in a graph with no cycles. A visual representation of $G_1^A$is shown in Figure 4.10, and the details regarding removing cycles are contained within Section 4.8.3. One graph contains 132 variable nodes, 80 factor nodes, and 208 edges, and requires approximately 0.54MB to store the graph in memory.

G_1^{KS} : the graphical representation of the first round of AES, but including the variable nodes representing the key scheduling. One graph contains 161 variable nodes, 129 factor nodes, and 350 edges, and requires $\approx$ 0.84MB to store the graph in memory.

[3] The key scheduling nodes are known to contain considerably less leakage relative to other intermediates

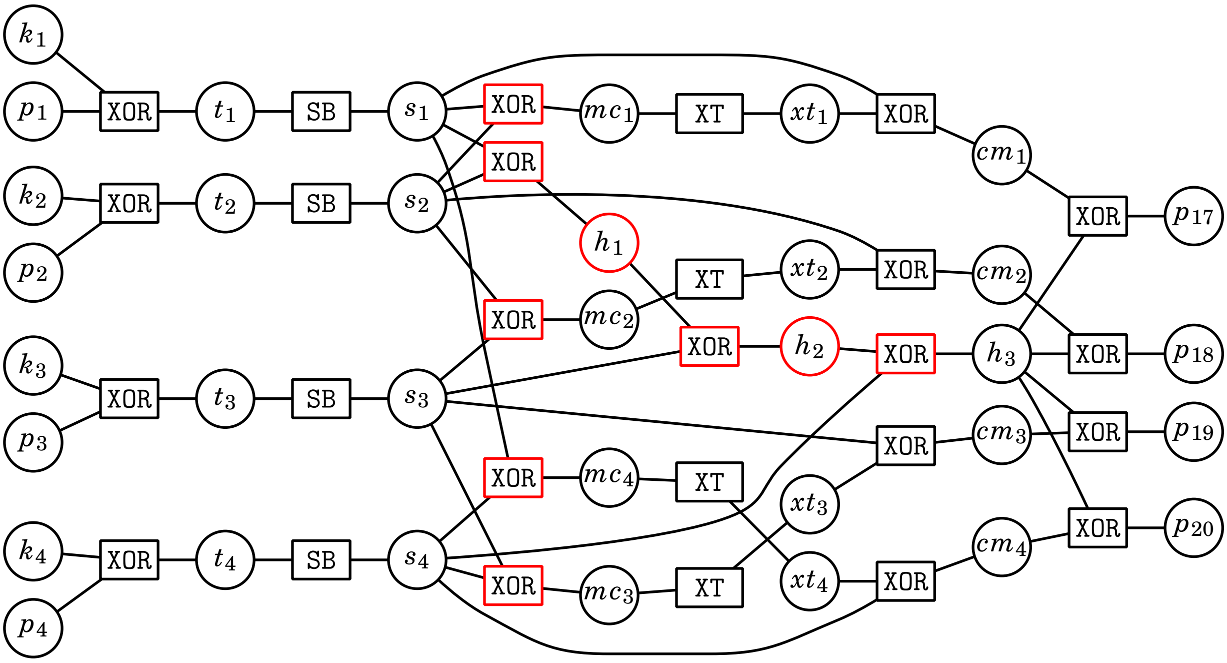


Figure 4.9: A factor graph representation of the first round of AES FURIOUS limited to one column, referred to as G_1; the red nodes are removed to generate G_1^A (as in Figure 4.10)

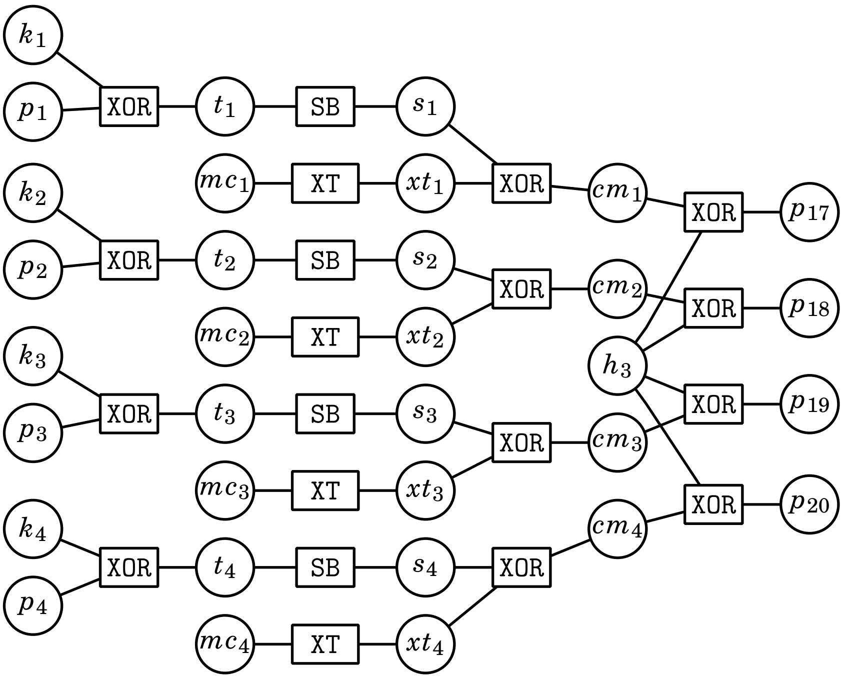


Figure 4.10: Factor graph G_1^A, created by removing selected nodes and edges from G_1 in order to remove cycles

4.7.7 Benefits of Method

In Section 4.7.1 we included a description of the attack proposed in *'Soft Analytical Side-Channel Attacks'* [7] by Veyrat-Charvillon et al. which required 33GB of memory to mount the Belief Propagation Attack against 5,000 traces. The proposal uses graph G. By reducing the graph to G_2, we maintain the attack success, whilst reducing the memory requirement down to approximately 4.5GB. The attack becomes feasible using this contribution.

This metric is also generalisable: depending on the target implementation (not necessarily AES, but any block cipher), the number of edges and nodes in the factor graph may change. In cases where the number of edges and nodes required in the factor graph is large, and memory is at a premium, one can employ the Hellinger distance metric to all nodes in the graph to detect unimportant nodes and remove them from the graph. This process can done automatically and without human supervision. Our experimental results show we can reduce the memory complexity by an order of magnitude.

4.8 Graph Convergence

4.8.1 Introduction

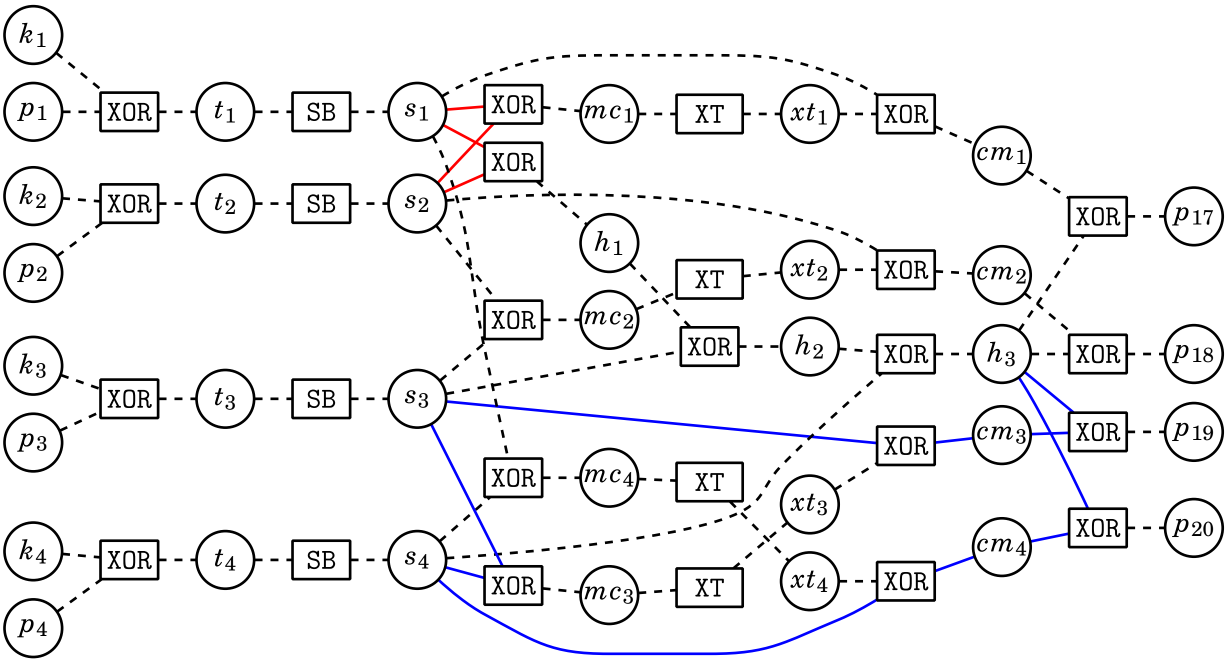


Figure 4.11: Copy of Figure 2.8: A factor graph representation of the first round of AES FURIOUS limited to one column, with two cycles highlighted: one small cycle in red, one larger cycle in blue (all other edges are dashed for visual aid). Note that Factor Graphs are inherently undirected.

Convergence in Belief Propagation is defined as meeting a stable equilibrium; when all the information in the graph has successfully propagated, and messages are no longer being updated after subsequent Belief Propagation iterations. When the graph is tree-like, convergence is guaranteed after a number of Belief Propagation iterations equal to the length of the longest path in the factor graph. When we perform Belief Propagation over a graph containing cycles, we refer to it as 'Loopy Belief Propagation'. Figure 4.11 shows the factor graph G_1, limited to one column of AES. This is an example of a graph with cycles: one is highlighted in red, and on is highlighted in blue. Many more exist within the graph. This means that when we apply the Belief Propagation algorithm to this graph, we are essentially performing 'Loopy Belief Propagation'.

4.8.2 Loopy BP

Identifying whether the Belief Propagation algorithm will converge is difficult when there are cycles present in the factor graph. As the number of cycles increases in the graph, this identification problem becomes exponentially harder, making it difficult to predict how the information will propagate over a cycle. Loopy BP has been studied in detail, such as in *'Loopy Belief Propagation for Approximate Inference: An Empirical Study'* [60] and in *'Generalized belief propagation'* [61]. In practice, our attack does not require convergence; we only require an accurate approximation of the marginals. Unfortunately, loopy Belief Propagation can often lead to chaotic behaviour [16], and it is unclear how one might extract an accurate approximation following this behaviour. We see this fluctuation in our work; Section 4.4.2 shows the results of the Epsilon Exhaustion technique. This checks when the graph has converged (or *almost* converged, using a specified threshold). If the graph does not terminate through this method, information is still being updated even after all information should have propagated. This is an example of the chaotic fluctuation first observed in *'Evidence of chaos in the Belief Propagation for LDPC codes'* [16].

Several solutions have been proposed that aim to extract an accurate approximation following chaotic fluctuation. One in particular involves taking the average marginal over a number of iterations, hoping that the fluctuation of information is close to an accurate result. However, if the information fluctuating is completely random (as has been observed in cases of extreme noise), this becomes a rather poor solution. We observe that it is the cyclic nature of the factor graph that is the underlying reason for lack of convergence of the BPA. From examination of the factor graph we recognise that there are a set of edges that, if removed, would remove the cycles from the graph. Therefore, we propose the hypothesis that eliminations of the cycles in a factor graph through the removal of certain edges may result in guaranteed convergence with minimal loss of data.

4.8.3 Acyclic Factor Graphs

To remove the cycles from the graph, we remove selected edges and nodes. By removing the nodes (and their connected edges) marked in red in Figure 4.9 we are left with the acyclic graph shown in Figure 4.10. Specifically, we remove the `XOR` nodes connecting the `SubBytes` output bytes to the mc nodes, as the experimental results using the Hellinger distance metric in Section 4.7 suggest the mc nodes provide a small amount of information relative to other nodes in the graph. Additionally, we remove the first two h nodes in each column: similarly, experimental result suggest these provide minimal information to the key bytes, and by removing them we produce an acyclic factor graph. Running the Belief Propagation algorithm over this acyclic graph will lead to guaranteed convergence, thus preventing the information fluctuation observed in cyclic graphs.

However, the consequence of removing nodes and edges comes with drawbacks. Most obviously, by removing nodes (in the case of Figure 4.9, the mc nodes as well as the h_1 and h_2 nodes) we remove any information that node would have provided. As a (rather extreme) example, suppose we have poor leakage on all nodes in the graph apart from the mc nodes. In other words, the leakage on all nodes is either erroneous or has a very low SNR, whereas the leakage on the mc nodes is perfect (the correct value has probability 1). In this scenario, running BPA on G_1 will yield a better final key rank than if we were to use the same leakage on G_1^A(as we have removed the only nodes that supplied accurate leakage).

One other drawback that comes when we remove edges is that we stunt the flow of information around the factor graph. By making the graph acyclic, we force information to only flow in one direction. This means that some nodes which were ‘close’ in proximity through the key bytes may now be further away, having to reroute through the only available path. In the following section we experiment to see the impact removing nodes has on the success rate of the Belief Propagation Attack.

4.8.4 Experimentation

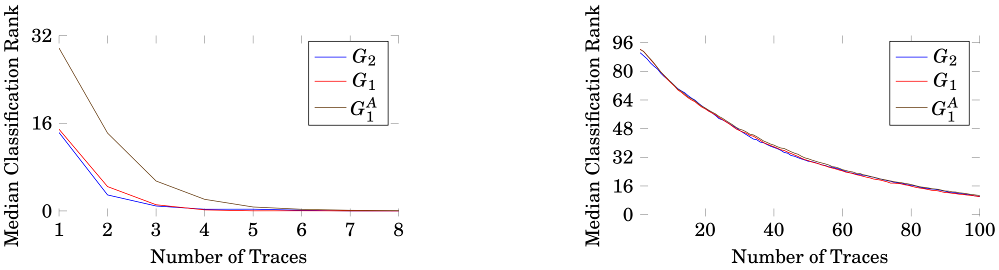
To experiment with the effect of removing cycles from the graph, we compare a standard Belief Propagation Attack on the two different graphs (cyclic and acyclic) with the same leakage information. We use the following graphs:

- The cyclic graph G_1 shown in Figure 4.9, created by removing second and further round nodes from graph G

- The cyclic graph G_2, created by removing nodes located in the third AES round onwards from graph G, resulting in the first two rounds of AES (shown in Section 4.7 to have identical success to G)

- The acyclic graph G_1^A shown in Figure 4.10, created by removing the mc nodes as well as the h_1 and h_2 nodes, shown in red in Figure 4.9

4.8.5 Results



(a) SNR = 2^{-1}

(b) SNR = 2^{-7}

Figure 4.12: Reduced Graph Comparison, comparing graphs G_2, G_1, and G_1^A with different SNRs

4.8.6 Observations

When the SNR is high (in this case, 2^{-1}) we see a small difference in the classification ranks when we compare the different graph structures. Graph G_2 has the best success, recovering the correct key in roughly 3 traces, whereas G_1 and G_1^A require 4 to 5 traces to recover the key. However, when the SNR is low (high noise, e.g. SNR = 2^{-7}), the difference between the graph structures is minimal.

4.8.7 Conclusion

Acyclic graphs provide guaranteed convergence, removing the 'loopy' approximation from the Belief Propagation algorithm. We therefore do not need to worry about implementing any 'early termination' techniques such as Epsilon Exhaustion, as G_1^A only needs 8 iterations of Belief Propagation to ensure that all information has fully propagated around the graph. Because of this advantage, we suggest making the graph acyclic when performing the attack in scenarios where the trace data has a low SNR. Our experiments show there is not a significant drop in the success rate when we remove the cycles. In our case, it was easy to remove cycles from our graph, as can be seen from Figure 4.9 to Figure 4.10. However, this many not always be the case; if the cycle removal does not seem trivial, it may require some experimentation to find the most efficient graph structure.

4.8.8 Benefits of Method

Convergence guarantees full propagation of all information in the factor graph. In an acyclic graph such as G_1^A, convergence is guaranteed in a constant number of BP iterations. This gives an upper bound on the runtime of the Belief Propagation Attack. Acyclic graphs also prevent the ‘chaotic’ fluctuation of data, observed in *‘Evidence of chaos in the Belief Propagation for LDPC codes’* [16]. Experimental results (Figure 4.12) show negligible difference in the attack success when the factor graph is acyclic. The benefits of this method are a faster runtime, a guarantee of convergence, with no observed decrease in success for noisy traces.

Chapter 5

Application to Observed Data

This chapter presents original work relating the application of the Belief Propagation Attack to traces taken from a physical device. I was responsible for writing the code, carrying out the experiments, analysing the results, and providing the write-up. This was all done under the supervision of Elisabeth Oswald.

5.1 Introduction

During the development of the Belief Propagation Attack, leakage data was required to test the effectiveness of the system. To acquire this data, we simulated the AES algorithm and provided the Hamming Weights of the intermediate values as the leakage information. We sampled noise from a Gaussian distribution to simulate the natural noise found when targeting real devices. The amount of noise chosen to be sampled varied according to the desired Signal to Noise ratio (SNR). This can vary depending on the device: in the paper titled *'ASCA, SASCA and DPA with Enumeration: Which One Beats the Other and When?'* [53], the SNR varies depending on the nature of the operation, ranging from 2^2 on `lpm` instructions (Load Program Memory) to 2^{-5} on `eor` instructions (XOR). For simplicity in our work, we opt to model all leakage using a constant SNR, irrespective of the instructions.

Having developed the Belief Propagation Attack using this simulated data, we now turn to a more accurate estimation of leakage data by using ELMO [17], as described in Section 2.4.3. ELMO provides coefficients for the weighted bit model, which accurately reflect the leakage of the M0. This method bridges the gap between using our simulated Hamming Weight model and leakage extracted from a real device. ELMO currently exists on the University of Bristol's Side Channel Groups GitHub page and another version is currently in development.

In this section, we document our move to running `AES FURIOUS` on a real device, extracting the leakage, and applying our Belief Propagation Attack to successfully recover the secret key. We improve on the state of the art by using a device that has a more complex leakage function to those used in other works [53], generating more noise and requiring inventive attack methods to successfully recover the key.

5.2 Practical Setup

To extract leakage from a device, we need access to the following:

- The target cryptographic device we wish to attack
- A power supply for the target device
- An oscilloscope to measure the power consumption
- Code to send data to the device, and to store the power consumption readings from the oscilloscope
- A clock generator, if the device's clock is unstable
- A Personal Computer (PC) to control all of the above

5.2.1 Target Device

The target device used was a SCALE Board [62] (**S**ide-**C**hannel **A**ttack **L**ab. **E**xercises). The board itself was designed by Daniel Page (my secondary supervisor) to facilitate the development of Side Channel Attacks. It hosts an ARM Cortex-M0. The ARM Cortex-M0 is a well characterised and understood processor, in the context of Side Channel Analysis. Previous work [31] has shown that its leakage function has linear terms as well as statistically significant second order terms (the leakage includes the number of bit flips in the registers), and that the noise is not significantly different for different instructions. It has a maximum clock frequency of 50MHz, which is identical to the Oscilloscope used in our attack setup. It runs `AES FURIOUS` [21] rewritten for the ARM Assembly language (originally written for the Atmel AVR).

It is expected that target devices have some means to communicate with a PC, so we are able to control the data being sent to and from the device. In the case of the SCALE board, the board hosts five SMA ports (coaxial RF connectors, three intended for input and two for output), for the following functions:

- *Power* input: to connect to an external power supply, although the board can also draw power from the USB-B port

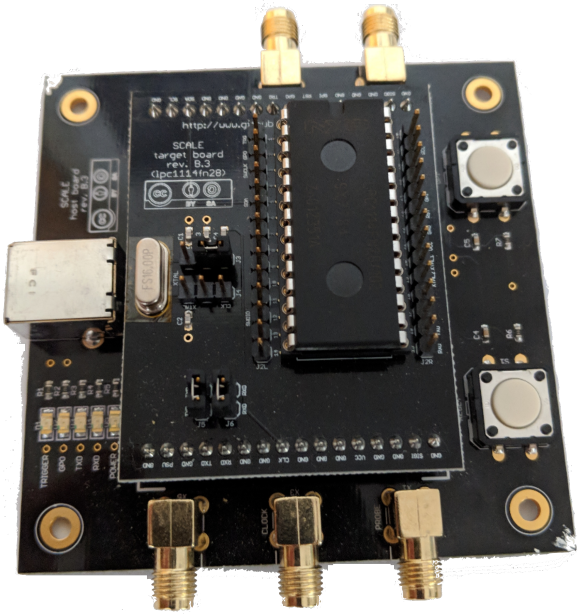


Figure 5.1: SCALE Board

- *Clock* input: to allow for an external clock, allowing for more accurate control than the on-board clock
- *Probe* input: used to facilitate fault attacks (unused for my work)
- *Trigger* output: a signal sent when the board begins or ends an encryption round; this sets the time frame in which the board sends its power measurements
- *Signal* output: the voltage over a resistor, captured by the oscilloscope and stored in the trace file

Unless using an external clock, we do not use the SMA inputs; we send data to the board using the USB-B port, and the oscilloscope records the measurements from the signal port when the trigger is toggled.

5.2.2 PC

The PC is connected to both the oscilloscope and the cryptographic device. It sends input data to the device via the USB-B port, and stores the power traces taken by the oscilloscope.

5.2.3 External Clock Generator

If the target device is small and cheap to manufacture, there is a chance it has an unstable clock. The clock jitter present on the device makes it difficult to extract meaningful traces, so we use an external clock to ensure stability. To ensure trace alignment, we use an external peripheral to maintain a stable clock. The clock generator we use is the Agilent 33250A, 80MHz Function, Arbitrary Waveform Generator. This piece of electronic test equipment is used to generate electrical wave-forms. By setting the clock speed to the exact frequency of the Scope (Section 5.2.4) we maximise our trace alignment accuracy.

Figure 5.2: Agilent Arbitrary Waveform Generator

5.2.4 Oscilloscope

The oscilloscope used was a PicoScope 2000 Series.There are several features that must be considered when using an oscilloscope:

- *Input Bandwidth*: this is the maximum bandwidth of the scope, and must be equal to or greater than the target device; in our case, our target device has a small clock bandwidth, so the input bandwidth of the scope was also small

- *Sampling Rate*: how many points of signal recorded per second, which must be greater than twice the most dominant frequency component of the power consumption signal (Nyquist Sampling Theorem [63])

- *Resolution*: this is the number of possible values after the analogue to digital conversion, most commonly 8 bits

The PicoScope has an 8 bit resolution at 500 MS/s, and runs at 50MHz.

Figure 5.3: PicoScope 2000

5.2.5 Acquisition Code

The acquisition script was written in `C` by Si Gao, modelled on the acquisition script for the SASEBO Board [64] (**S**ide-channel **A**ttack **S**tandard **E**valuation **BO**ard). This script generates the uniformly random plaintexts and the random (or fixed) key, sends them to the device, and sets up the trigger on the PicoScope to start recording power values when the device starts the encryption method. Once encryption is over, the code stores the meta-data and the trace data using RISCURE's `.trs` file format used in their own Inspector Tool [65]. In order to ensure the trigger starts and ends correctly, an overhead of 0.5 seconds ensures we capture all significant Points of Interest (as described in Section 2.3.4). As a consequence, running the algorithm takes roughly 0.5 seconds per trace.

5.2.6 Acquired Traces

For the majority of our trace sets (unless stated otherwise), we generated 210,000 traces of G_2, each with 51,250 samples. We chose to use this number of traces in order to build templates that could achieve first order success. At 0.5 seconds a trace, this takes just over a day to acquire, and requires roughly 22GB. All these traces were stored on an external hard drive, and backed up on two machines.

5.3 Parsing the Tracefile

5.3.1 RISCURE `.trs` format

The '.trs' format is a proprietary encoding owned by RISCURE. The encoding consists of a header block, followed by the trace block. In the header block, each object value is preceded by a tag field (and in some cases a length field). Table 5.1 shows the various header tags along with a

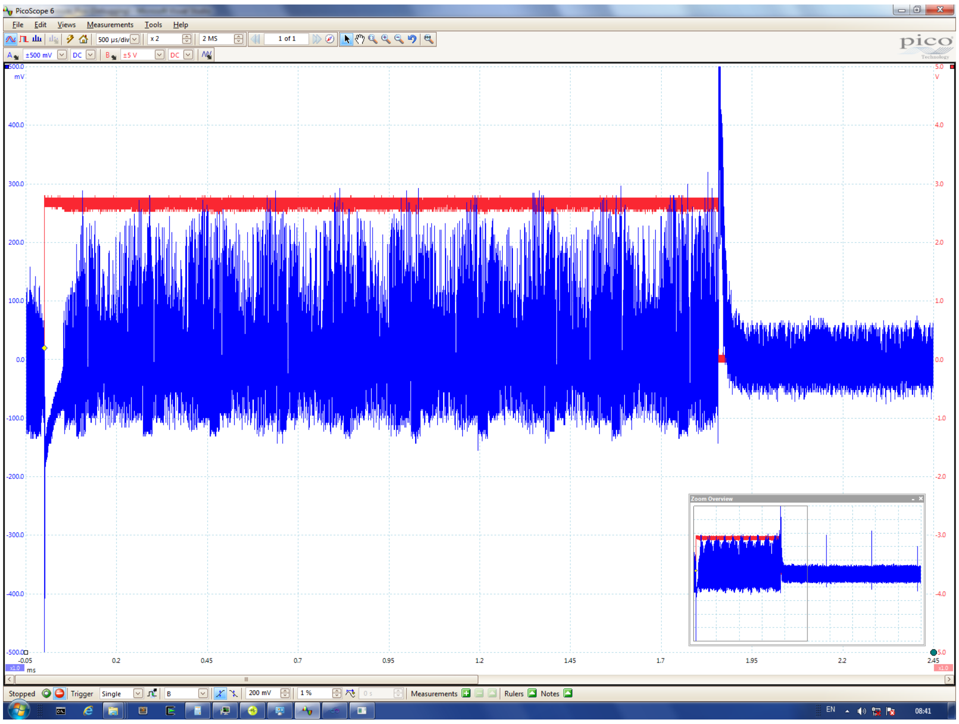


Figure 5.4: A Screenshot of a single trace, measured on the PicoScope

description of each. After the header block, each trace is stored along with the key / plaintext pair used to generate the trace.

Tag	Name	Type	Length	Meaning
0x41	NT	int	4	Number of Traces
0x42	NS	int	4	Number of Samples per Trace
0x43	SC	byte	1	Sample Coding
0x44	DS	short	2	Length of cryptographic data included in trace
0x5F	TB	none	0	Trace block marker: an empty TLV that marks the end of the header

Table 5.1: Table of `.trs` header tags along with their descriptions

5.3.2 Our Tracefile

The trace data we acquire consists of 210,000 traces: 200,000 traces where the key is random, and 10,000 traces where the key is fixed to have the following value (chosen arbitrarily):

```
[0x54, 0x68, 0x61, 0x74, 0x73, 0x20, 0x6D, 0x79,
0x20, 0x4B, 0x75, 0x6E, 0x67, 0x20, 0x46, 0x75 ]
```

In order to use the acquired trace data, we must extract the leakage that will aid us in the attack. The Python file `correlation.py` in the project performs the following steps to reformat the trace data into files that facilitate the attack. The construction of the factor graph is fixed; that is, we know we are using AES FURIOUS and, as such, we construct the factor graph from our knowledge of this algorithm. If one wishes to apply our code to a different cryptographic algorithm (e.g. DES) then one would need to manually code the factor graph and leakage simulation for that implementation. Therefore, when we refer to a 'variable node' in the following sections, this is because we know which variable nodes will be included in the factor graph representation of the acquired traces.

5.3.3 Separating the Data

The .trs file was designed to be read easily by the RISCURE Inspector Tool. It is simple to parse, providing us with the following information:

- Number of traces
- Number of samples per trace
- The sample space (the data types and sizes of the power values)
- The plaintext and key for each trace, along with the trace data

We extract the meta data in the header, and store the trace data in a numpy (mathematical library for Python) file for ease of access. Due to the large size of the trace data, we must memory map this file: this keeps the trace data on the hard disk when reading and writing, avoiding loading it into memory at the expense of slower reading and writing.

5.3.4 Computing Extra Data

After extracting the key and plaintext for each trace, we are able to generate all intermediate values. To do this, for each key and plaintext pair in the trace set, we run AES in Python, storing all intermediate variables when they are computed (this is handled by the leakageSimulatorAESFurious.py file). We then store these intermediate values in files to be accessed by other functions.

5.3.5 Points of Interest Detection

Once we have all intermediate values, we can use correlation analysis to find the Points of Interest, as described in Section 2.3.4. For each variable node, we find the correlation value (using numpy.corrcoef) of the values of the node with each sample point across all traces.

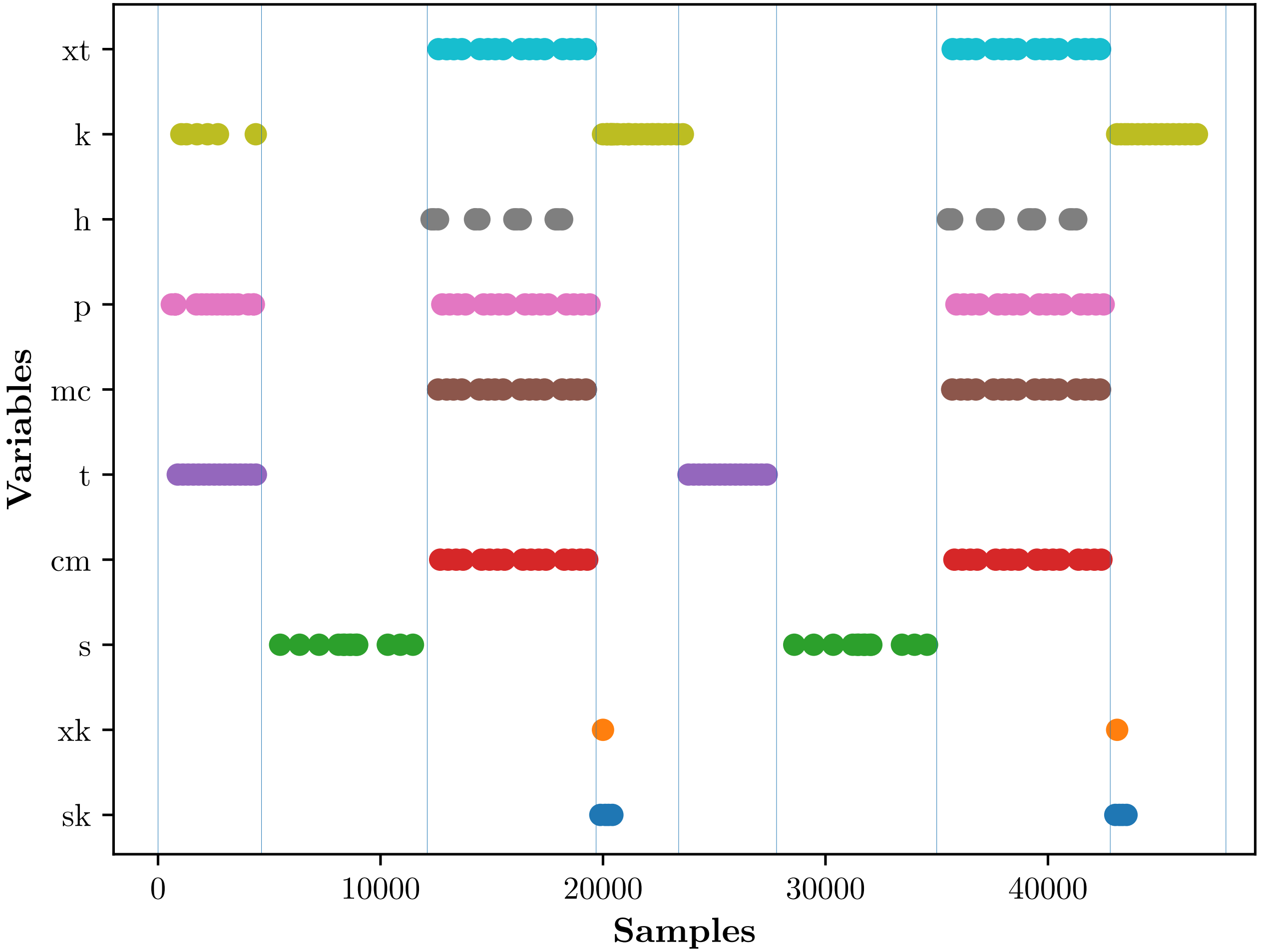


Figure 5.5: Hamming Weight Based Correlation Analysis

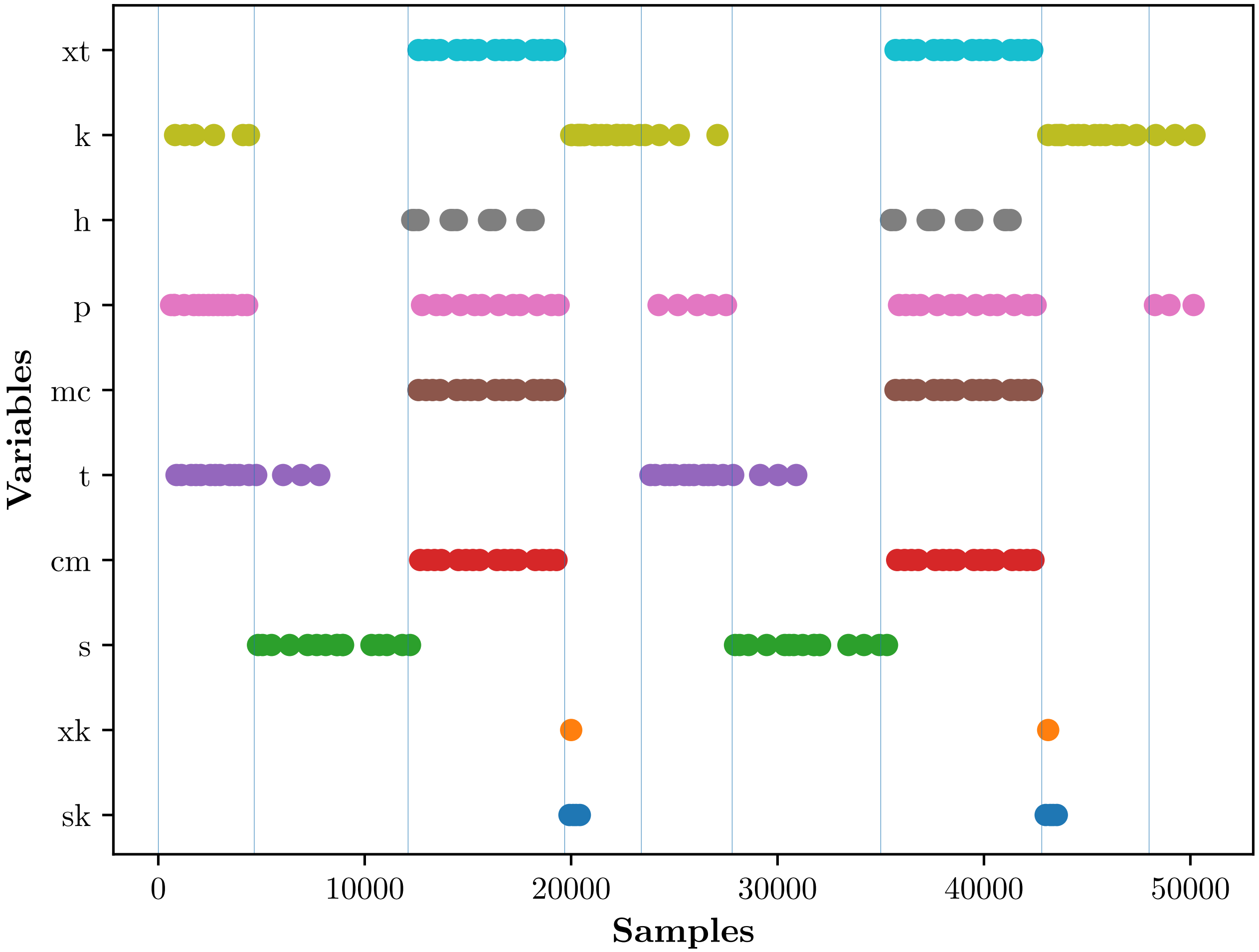


Figure 5.6: Identity Based Correlation Analysis

Figure 5.6 shows the top correlated timepoint when using the identity values of each node for the correlation analysis. Figure 5.5 shows the top correlated timepoint when using the *Hamming Weight* of the identity value. For illustration purposes, there are vertical lines separating the 'sections' of AES (e.g. `AddRoundKey`, `SubBytes`, `MixColumns`, and `KeyExpansion`). We can see from these plots that the Points of Interest chosen using the Hamming Weight of the identity value are clustered close together, whereas the points selected using the identity value are often not contiguous. For this reason, we opt to use the timepoints found through correlating the Hamming Weights.

The Point of Interest detection step is a lengthy procedure, and is the most computationally intensive method of `correlation.py`. After we have performed the detection step on all samples within the trace (each producing a correlation coefficient), we store the list of coefficients in a file, along with the highest correlating time point, and the power values associated at this time point.

5.3.6 Final Layout

The directory list is illustrated in Figure 5.7.

```
📂 TRACES
├ tracefile.trs
└ 📂 tracefile
   ├ metadata.pkl
   ├ musigma.dict
   ├ 📂 coefficientarrays
   │  └ ...
   ├ 📂 keys
   │  ├ keys.npy
   │  └ extrakeys.npy
   ├ 📂 plaintexts
   │  ├ plaintexts.npy
   │  └ extraplaintexts.npy
   ├ 📂 tracedata
   │  ├ tracedata.npy
   │  └ extratracedata.npy
   ├ 📂 time points
   │  └ ...
   ├ 📂 powervalues
   │  └ ...
   └ 📂 realvalues
      └ ...
```

Figure 5.7: Directory Listing for the parsed `.trs` file

Note that when a file is prefixed by `extra`, it refers to the 10,000 traces using the fixed key. It is these that we attack to test the effectiveness of the Belief Propagation Attack.

5.3.7 Univariate Templating

For each variable node, we sort the traces into n sets, where n is equal to the number of possible values: $n = 256$ when using the identity value (as all variables are bytes), or $n = 9$ when using the Hamming Weight model (as variables are 8 bits, they can have the Hamming Weight values 0 to 8 inclusively). For each of these sets, we compute the mean and standard deviation pair of power values, giving us the list of templates:

$$[(\mu_0, \sigma_0), (\mu_1, \sigma_1), \ldots, (\mu_{n-1}, \sigma_{n-1})]$$

These templates will be used to perform a template matching step as described in Section 2.3.5.

5.3.8 Applying the Belief Propagation Attack

Now that we have our trace data formatted in a simple way, we can use the leakage data in the Belief Propagation Attack. After building our factor graph, we template match the corresponding power value (found in `powervalues/`) with the node's templates (found in `musigma.dict`) for each trace. This gives us a 'likelihood array' which, when normalised, becomes the initial distribution for that node for the specific trace.

5.4 Real Data vs Simulated Data

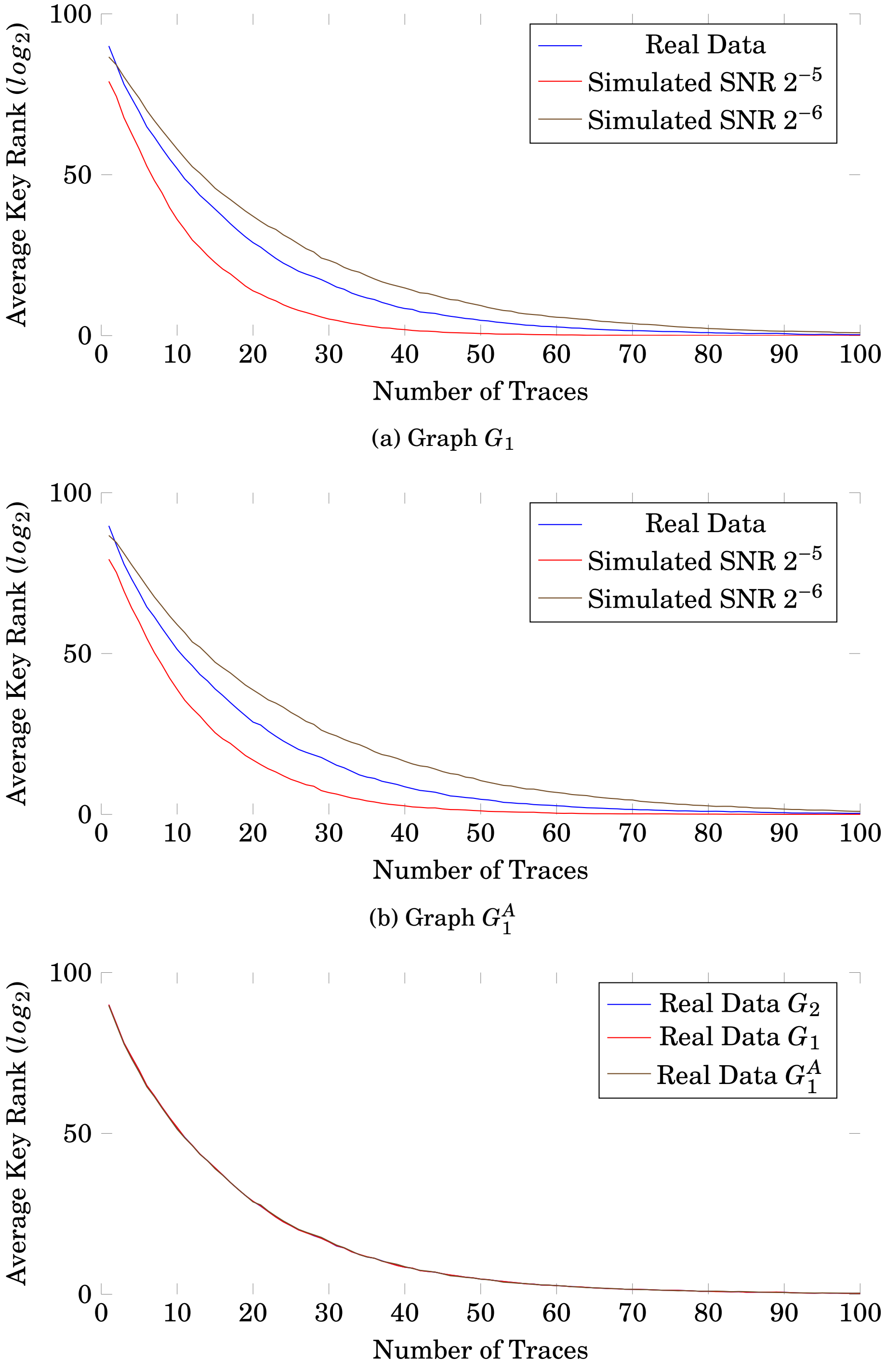
The results shown in Chapter 4 used the simulated ELMO data as the attack target. In order to validate the various improvements proposed in this chapter, we must confirm that our attack yields similar results on leakage data taken from a real device. Previous work has shown that the SNR of the leakage taken from a device is dependent on the type of instructions [31], but for simplification one can model the overall leakage with a constant SNR.

In this Section, we run the Belief Propagation Attack on both the simulated data (using the ELMO weighted bit model with Gaussian noise using an appropriate SNR) and the real data taken from the ARM Cortex-M0. We infer the SNR by running BPA with a number of different SNR values, and we selected the SNR that gave the most similar results to the attack on the ARM Cortex-M0 leakage.

5.4.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100
   -raes 1 [--RM_C, <none>] [--REAL, -snrexp -5, -snrexp -6]
```



(a) Graph G_1

(b) Graph G_1^A

(c) Real Data Graph Comparison

Figure 5.8: Plots showing Belief Propagation Attack results on simulated data (using an SNRs of 2^{-5} and 2^{-6}) and on trace data taken from the ARM Cortex-M0, using cyclic graph G_1 and acyclic graph G_1^A

5.4.2 Observations

Figure 5.8a compares a Belief Propagation Attack against the trace data taken from the ARM Cortex-M0 to an attack against simulated data, using graph G_1. We see that the average key rank of the real trace data is bounded in between the SNRs 2^{-5} and 2^{-6}. This result is echoed in Figure 5.8b where the same attack is mounted against graph G_1^A.

Figure 5.8c compares the Belief Propagation Attack results using the real data on different graphs. The experimental results on G_1 and G_1^A are identical, echoing results found using simulated date in a high noise scenario in Section 4.8.3.

5.4.3 Conclusions

Our real trace data has an observed SNR between 2^{-5} and 2^{-6}. There is more noise present on this device than the device used in *'Soft Analytical Side-Channel Attacks'* [7], where Veyrat-Charvillon et al. observed SNRs ranging from 2^2 to 2^{-5}. We confirm that our device uses a more challenging leakage model (harder to attack).

Our experimental results show that an attack using the acyclic graph G_1^A yields identical results to attacks using the cyclic graphs G_1 and G_2. This is not surprising: we saw the same results using simulated data with an SNR of 2^{-7} (see Section 4.8.3). The benefits that come with using the acyclic graph G_1^A (as described in Section 4.8) include guaranteed convergence, along with a bound on the number of required BP iterations. For this reason, we propose to use graph G_1^A when attacking the `AES FURIOUS` implementation.

5.5 Linear Discriminant Analysis

We describe Linear Discriminant Analysis (LDA) in Section 2.6.1. We can apply this statistical technique to our template building scenario. The LDA Classifier predicts the value of the intermediate variable, having been provided with a window of power values over the time point where the intermediate variable is computed and loaded into memory.

5.5.1 Implementation

To implement LDA we use the `scikit` package, a third-party python library that comes with an LDA class, which is able to handle the training and predictions of an LDA classifier. We train a separate LDA classifier for each variable node in the graph. Due to the size of the full AES graph G, we opt to reduce this to G_2, following the conclusions made in Section 4.7. With this reduction, we reduce the number of required trained classifiers from 1212 to 188. We train with 190,000 training traces, each using a random key and random plaintext (taken from our set of 210,000 available traces). Training an LDA classifier is straightforward, and is demonstrated in Listing 5.1.

Listing 5.1: Python code to train an LDA Classifier

```
# Get training labels for variable index i
# real_values [ variable_index, trace_number ]
y = real_values[i, :traces-validation_traces]
# Get training data window according to timepoint tp and window w
# trace_data [ traces, samples ]
X = trace_data[:-validation_traces, tp[i] - (w/2):tp[i] + (w/2)]
# Set up linDisAnalysis
lda = linDisAnalysis()
lda.fit(X, y)
# Save
pickle.dump(lda, ...)
```

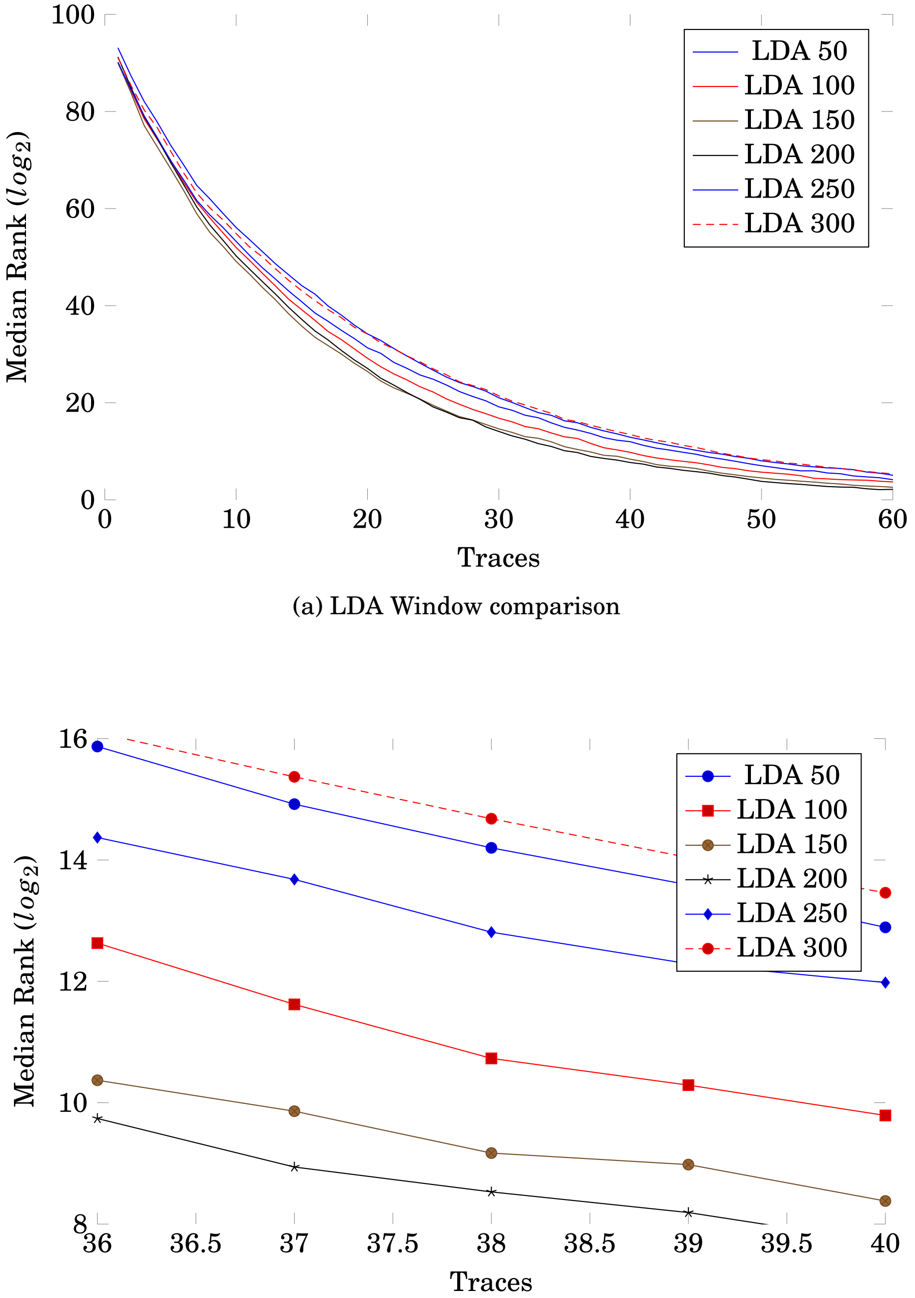
5.5.2 Optimal Window

Now that we are considering a multivariate templating method, we firstly need to find the optimal window of power values to provide to the LDA classifier. Too small, and we do not provide enough information for the classifier to 'learn' the characterisation of the leakage. Too large, and we run the risk of including redundant information that the classifier must sift through in order to find the significant leakage. To compare the window results, we first trained LDA classifiers for all variables in G_1 using different window sizes, ranging from 50 to 300. We ran the Belief Propagation Attack using the LDA classifiers for each window size, plotting the mean rank ($\log_2$) of the final key.

5.5.2.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100 -raes 1
                --REAL --LDA -tprange [2, 5, 10, ..., 250, 300]
```



(a) LDA Window comparison

(b) LDA Window comparison closeup from 36 to 40 traces

Figure 5.9: Comparing the window size parameter for the LDA Classifier

5.5.2.2 Conclusions

Figure 5.9a shows that the variance in performance based on windows in the range 50 to 300 is small. When we look at a closeup (as shown in Figure 5.9b), we see that the best result uses a window of 200 values (100 samples either side of the Point of Interest). The experimental results suggest that a window of 200 is locally optimal, as it outperforms windows on either side (150 and 250).

5.5.3 Comparison to Gaussian Templates

In order to show the benefits of using LDA over univariate templating, we run the Belief Propagation Attack using each of the classifiers independently, comparing the final distribution of the key bytes.

5.5.3.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100 -raes 2
                 --REAL [<none>, --LDA]
```

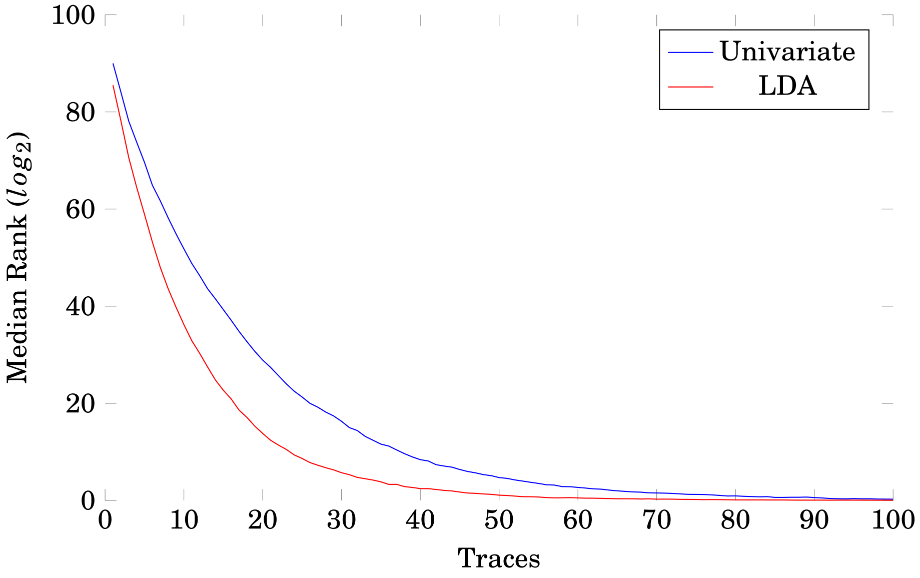


Figure 5.10: A comparison of Belief Propagation Attacks using Gaussian Univariate Templating against using Linear Discriminant Analysis as a classifier

Figure 5.10 compares the Belief Propagation Attack using univariate templating and using an LDA classifier. LDA outperforms the univariate method by successfully recovering the key after 60 traces, whereas the univariate method achieves first order success at around 90 traces.

5.5.3.2 Conclusions

LDA is able to outperform the standard univariate templating method by utilising a window of leakage over the target time point. Although the LDA classifier requires setup time in the offline phase, this timing overhead is not present during the attack phase. However, LDA has its own limitations: *'Enhancing Dimensionality Reduction Methods for Side-Channel Attacks'* by Cagli et al. [66] show that LDA is often overlooked due to its practical constraints, as it requires the number of traces to be larger than the dimension (size) of them (known as the *Small Sample Size Problem*). This in itself has been studied extensively in literature, with proposals for adjustments to the LDA algorithm that are able to overcome the Small Sample Size Problem for LDA [67].

We have shown that the multivariate technique provides improvement over the univariate method. We now turn to a much more powerful tool utilised in classification problems: Neural Networks.

Chapter 6

Application of Neural Networks for the BP Attack

This chapter is concerned with the implementation and experimentation of Neural Networks as a leakage classifier, and the application of the Neural Networks to the Belief Propagation Attack. I was responsible for writing the code, carrying out the experiments, analysing the results, and providing the write-up. This was all done under the supervision of Elisabeth Oswald. A subset of the content included in this chapter has been submitted to CT-RSA 2020, with the title *Not a Free Lunch but a Cheap Lunch: Experimental Results for Training Many Neural Nets*.

6.1 Introduction

The aim of this chapter is to utilise the power of Deep Learning classification by training a Neural Network to classify power leakage. In Section 5.5 we show how we can use Linear Discriminant Analysis as a power value classifier, and our experimental results show an improvement over the univariate Gaussian templating method.

Our hypothesis is that when we combine the Neural Network classification method to the *profiling* phase of our Belief Propagation Attack, we will improve the attack success over both the univariate Gaussian templating method and the Linear Discriminant Analysis classifier. Our reasoning behind this is that Neural Networks are widely used in the fields of image classification and voice recognition. They excel at classification tasks, and our use case requires us to classify the time series of power values. We are not the first to apply Neural Networks to Side Channel Analysis, but we are the first to consider maximising the *per trace* classification result, as well as training multiple Neural Networks to target multiple leakage points independently from each other, then combining the results using Belief Propagation (as will be discussed within this

Chapter).

6.1.1 Deep Learning in Side Channel Analysis

The rise of papers involving Deep Learning assisted Side Channel Attacks has been noticed by academics and industry professionals alike. Some of the more recent papers use AES as an attack target, all using their own preprocessing and attack methods; even the choice of device varies, including the countermeasures included on the target implementation. However, there has yet to be a paper that targets multiple intermediates within the attack trace; the other papers solely look at a single leakage point, such as the SubBytes output. In addition, the devices that have been targeted have simple leakage models, with an easy-to-target signal accompanied by little noise.

Just like the Template Attack method (described in Section 2.3.5), the Neural Network assisted attacks are split up into two phases; the *offline* phase, where the target device is profiled using a replica of the target, and the *offline* phase, where the target is attacked directly, by using the previously built profiles. In the offline phase, we acquire leakage information from this replica; these traces use known plaintexts and keys. This leakage information becomes the training set, which we explore in Section 6.2.2. At this point, we select a model for the Neural Network. The structure itself is usually based off a model that has been used to do something similar [10]. The hyperparameters for this network also need to be chosen, with respect to the training data [15]. When the network is trained using the training data, the network will be able to parse unseen leakage information, and produce some probability distribution over the key space; this is part of the online attack phase.

One of the clear advantages of using Deep Learning is the Point of Interest detection; classical methods (such as univariate templating) require a timepoint to be chosen in advance before templates can be made. A feature of Neural Networks is the automatic Point of Interest Detection; by using a loss function, the Neural Network selects the features within the dataset that result in the best classification success. By providing a large window of leakage information, the Neural Networks will learn to select the appropriate leakage values without the need of human interference.

6.1.2 The 'No Free Lunch' Theorem

The "no free lunch" theorem was posited by David Wolpert and William Macready in 1997 [68]. This states that "any two optimisation algorithms are equivalent when their performance is averaged across all possible problems". This is relevant to Neural Networks, and their general performance in different scenarios. Research has shown that a network that has been trained effectively for one use case does not necessarily perform well when used to solve a different problem [11]. In fact, for supervised learning, there is no perfect learning algorithm (that provides the best result for all classification problems) [69].

Our use case for the Neural Networks is in combination with an Inference Based Attack, in which we combine information from multiple intermediates to recover the key. We wish to discover whether the “no free lunch” theorem applies in this context: our hypothesis is that there is no network structure that performs best for all of the leaking intermediates, despite all being part of the same trace (i.e. the “no free lunch” theorem is applicable in the context of power leakage classification).

6.2 Implementation

We wish to use Neural Networks to classify leakage from different intermediate variables. When we provide a window of leakage information to the network, the network should return a probability distribution over the target intermediate’s value space. This probability distribution can then be used to attack the device, either in a profiled template attack scenario or the Belief Propagation Attack from Chapter 4.

Neural Networks are defined in Section 2.6.2. There are many choices to be made when selecting a network for a certain task. Results presented in *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* [15] show that a Multi-Layer Perceptron (defined in Section 2.6.4.2) excels at leakage classification when there is no clock jitter in the traces. We shall use this as a starting point. One must also consider the various hyperparameters to be chosen, which are highlighted in Section 2.6.5.

To implement the Neural Networks, we used the Keras API [70], using TensorFlow as a backend (identical to [15]).

6.2.1 Keras

From the Keras webpage [70]:

Keras is a high-level Neural Networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

Keras is user friendly, providing powerful tools to shape and tune the Neural Network according to our specification. To train and test a model in Keras, we use the Python files `train_models.py` and `test_models.py`, briefly described in Section 4.2.8 and Section 4.2.9. These files were adapted from the ASCAD code hosted on Github [18] from files of a similar name (`ASCAD_train_models.py` and `ASCAD_test_models.py`).

In these files, Keras is used to build the model in an intuitive manner. Listing 6.1 demonstrates how simple it is to build a Multi-Layer Perceptron using the Keras library.

Listing 6.1: Python function to build a simple MLP

```
# Function to build and return an MLP
def mlp(nodes_per_layer, number_of_hidden_layers, input_length, classes,
    ↪ loss_function, learning_rate):
  # keras.models.Sequential initialises a standard model
  model = Sequential()
  # Add the Input Layer
  model.add(Dense(mlp_nodes, input_dim=input_length, activation='relu'))
  # Add the Hidden Layers
  for i in range(number_of_hidden_layers):
      model.add(Dense(nodes_per_layer, activation='relu'))
  # Add the Output Layer
  model.add(Dense(classes, activation='softmax'))
  # Initialise the optimiser (we use RMSProp here)
  optimizer = RMSprop(lr=learning_rate)
  # Compile the model with respect to the loss function and optimiser
  model.compile(loss=loss_function, optimizer=optimizer, metrics=['accuracy'])
  # Return the compiled MLP
  return model
```

Training a model is done in a similar fashion to LDA, as shown in Listing 6.2; we only need to fit the model to the training data and labels after reshaping them. The training and validation labels must be ‘one hot encoded’ (converted to a vector of size 256, where all values are 0 except for a 1 at the index of the correct label), which is as simple as calling the `to_categorical()` function from the `keras.utils` package on a vector of identity values (the real byte values as stored in the registers during the computation of the target algorithm).

Listing 6.2: Python code to train a model with some training data

```
# Training a model with the training data and labels, and validating with
    ↪ separate data
model.fit(x=training_data, y=training_labels, batch_size=batch_size, epochs=
    ↪ epochs, callbacks=callbacks, validation_data=(validation_data,
    ↪ validation_labels))
```

Once trained, we can test by extracting unknown power values from the testing set (along with their corresponding labels), and querying the model, as seen in Listing 6.3. We can compare the output of this query (a probability distribution) to the real identity of the value using whatever metric we choose (which we discuss later in this chapter).

Listing 6.3: Python code to query a model with some testing data

```
probability_distribution = model.predict(power_values)[0]
```

We used TensorFlow as a back end for Keras, as described in Section 2.6.8.

6.2.2 Training

The first step when training a network (after choosing a network structure) is to generate training data and training labels, as described in Section 2.6.5. For our specific use case, the training data will take the form of a large window of leakage data (power values). The training labels will be the identity value of the target intermediate, where the point of interest lies somewhere within the provided window.

We use 190,000 traces in our training data set, each using a random[1] key and a random plaintext (identical to the traces provided to the LDA classifier).

6.2.3 Validating

The validation data is then used to check how well the model is learning by using unseen data. The specified loss function is used here; in our case, we use categorical cross entropy (which will be defined in Equation 6.5). We utilise open source tools to visualise this validation, such as TensorBoard (see Section 2.6.9). By using visualisation methods, we can check to see whether the model is overfitting early, or whether the model is learning anything at all.

The data used to validate the Neural Network also uses random keys and random plaintexts. We do not used a fixed key here as this would prevent the network from learning the effects of using different key values. We use 10,000 of these traces.

6.2.4 Testing

The testing data uses random plaintexts, but a fixed key. This was done for multiple reasons; firstly, by having a fixed key, we allow ourselves the possibility to test the networks using a Differential Power Analysis attack, which would only work if the key remained constant for a number of traces. Secondly, the work presented in *'A Stochastic Model for Differential Side Channel Cryptanalysis'* [71] shows that the AES `SubBytes` operation has *equal images under different subkeys*, often referred to as having the EIS property. Therefore, we are able to make assumptions on other keys when we use a fixed key to test, as they should all act in an identical manner.

It is often the case that cross validation is used during the testing stage, as is the case in *'Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database'* [15]. Our work differs in that we are interested in the classification performance per

[1] Random in this context can be interpreted as 'non-fixed'

trace. Therefore, cross validation is not necessary. Instead we use the holdout method (training and validating split into two separate sections).

6.2.5 Success metrics

We define vector d to be a probability distribution produced by the network after classifying some trace for an unknown subkey $k \in \mathcal{K}$, which has correct subkey value k^*. The probability given to k^* from d is denoted as $d[k^*]$.

We first define the 'rank' of k^* given a trace T in Equation 6.1. The rank is an integer between 1 (the best and largest probability relative to others) and $|\mathcal{K}|$ (the worst).

$$\text{rank}(k^*, T) = |\{k \in \mathcal{K} | d[k] > d[k^*]\}| \tag{6.1}$$

Knowing the rank based on a single trace does not help us a great deal: the rank is derived from input data that may be regarded as random, such that the 'rank' acts as a random variable. Instead, we compute the rank on a set of traces $\mathbf{T}$, and we look at both the median rank and the median probability, defined in Equations 6.2 and 6.3 respectively.

$$\text{medianRank}(k^*, \mathbf{T}) = \underset{T \in \mathbf{T}}{\text{median}}(\text{rank}(k^*, T)) \tag{6.2}$$

$$\text{medianProbability}(k^*, \mathbf{T}) = \underset{T \in \mathbf{T}}{\text{median}}(d[k^*]) \tag{6.3}$$

We need to provide the correct subkey value k^* to the Neural Network in the form of a training label. We use one-hot encoding to do this; k^* is represented as a sparse vector, with a single 1 at index k^*, as defined in Equation 6.4.

$$\text{oneHot}(k^*) = (\underset{0}{0}, \ldots, \underset{k^*}{1}, \ldots, \underset{255}{0}) \tag{6.4}$$

Therefore, the loss function 'categorical cross entropy' can be defined in Equation 6.5.

$$\text{crossEntropy}(d, k^*) = -\sum_{i=0}^{255} y_i \log d[i] = -\log(d[k^*]), \text{ where } y_i = \begin{cases} 1 & \text{if } i = k^*, \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

6.3 Analysis of the ASCAD Database

In 2018, Prouff et al. published the paper titled *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* [15], in which the authors included a detailed study on applying deep learning to Side Channel Analysis. This wasn't the first paper of its kind, but it provided a welcome step into how one must carefully structure the network and select hyperparameters in order to maximise the effectiveness of the attack. They target AES as an example, using an 8-bit device that leaks in such a way that it is straightforward to infer the Hamming Weight of the intermediate values.

There were two main contributions in their paper that we focus on in this work; the first is the detailed discussion they provide regarding the choice of hyperparameters to optimise the classification of power leakage, and the second is the comparison between Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) in the context of power leakage classification on jittery traces. In their work, they conclude that the MLP outperforms the CNN when there is minimal clock jitter, but the CNN is more resilient against trace misalignment. In this work, we do not aim to defeat hiding countermeasures (random delays that cause clock jitter), but instead we focus on whether it is possible to build networks for all intermediates efficiently. Therefore we focus on MLPs for our intermediate classification, rather than the more complex CNNs. It is interesting to note here that in the jitter free scenario, the template attack Prouff et al. implemented outperformed both the MLP and the CNN.

Once a model structure has been chosen (in this case, the MLP), the next step is to select the hyperparameters for the network. The ASCAD paper splits the hyperparameters up into two groups: *training* parameters, such as how long to train the model for (number of epochs) and the size of the training set, and *architecture* parameters, such as how big the model is in terms of hidden layers and neurons per layer. These hyperparameters are selected one or two at a time, and different values are tested using 10-fold cross validation with different sizes of the data set. The results are compared using some metric (in their case, the mean rank w.r.t the number of test traces), and the best value is saved, then used as the default hyperparameter value when testing other hyperparameters. The training set Prouff et al. use was partitioned into 50,000 profiling traces and 10,000 validation traces (both with random[2] keys and random plaintexts).

6.3.1 ASCAD's Choice of Hyperparameters

Out of the two groups of hyperparameters, the training parameters were the first to be selected, shown in Table 6.1. The architecture parameters are shown in Table 6.2. These tables list the order in which the training and architecture parameters were chosen (with the exception of the batch size and the number of epochs, which were tested and selected at the same time). In all cases, a probability distribution was required as the network output, so the output layer was always chosen to be Softmax (which converts the output into a probability distribution). Similarly, the loss function was chosen to be categorical cross entropy throughout the testing.

Unfortunately, it is not clear from the paper whether the performance figures included always target the leakage from the `SubBytes` output, but we assumed this to be the case. With this

[2] Random sampling from a uniform distribution

assumption, we summarise that the MLP outperforms the CNN, and the MLP reaches a stable first order success after 300 traces.

Table 6.1: The tested values and best values for chosen training parameters for the MLP in the ASCAD paper.

Parameter	Tested Values	Best Value
Size of Training Set	10,000, 20,000, 30,000, 50,000, 70,000, 90,000	50,000
Number of Epochs	100, 200, 400	200
Batch Size	50, 100, 200, 500	100
Learning Rate	10^{-7}, 10^{-6}, ..., 10^{-3}	10^{-5}
Optimiser	Adadeita, Adagrad, Adam, RMSProp, SGD	RMSProp

Table 6.2: The tested values and best values for chosen architecture parameters for the MLP in the ASCAD paper.

Parameter	Tested Values	Best Value
Number of Hidden Layers	3, 4, 5, 6, 7, 8, 11	4
Number of Nodes per Hidden Layer	20, 50, 100, 150, 200, 250, 300, 500	200
Activation Functions	Hard Sigmoid, Linear, ReLU, Sigmoid, Softmax, Softplus, Softsign, Tanh	ReLU

6.3.2 Reusing the ASCAD MLP for the M0 data

Continuing from Section 5.5.3.2, we decided to use Neural Networks to classify our leakage taken from the SCALE board, which hosts an ARM Cortex-M0. Although architecturally the AES implementations running on both were similar, the M0 is a more complex processor than the AVR ATMega8515 used in *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* [15], as the M0 processor features a range of leakage functions. Our idea was to see how well the Neural Networks used for the ASCAD dataset performed on our M0 data. Our hypothesis was that, at least for the `SubBytes` output, the results on our data would be similar to those in the ASCAD paper.

Table 6.3 shows the classification results using the following classification models on the M0 data:

- A uniform distribution, included as a worst-case benchmark (all values predicted with equal probability)

- A pretrained CNN and a pretrained MLP, both originally trained on the ASCAD database, then retrained on the M0 data; this retraining technique is often used in image classification
- A CNN and an MLP that were trained solely on the M0 data
  - Both the pretrained models and the models only trained on the M0 data used the optimal hyperparameters found in the ASCAD paper (they are referred to as MLP_{best} and CNN_{best})
- A univariate model, as described in Section 2.3.5
- A Linear Discriminant Analysis classifier, as described in Section 5.5

Table 6.3: Classification results using different learning algorithms, attacking the first `SubBytes` output byte

Classifier	Median Rank	Median Probability	Mean Probability
Uniform	128	0.003906	0.003906
CNN, Pretrained	127	0.001150	0.003813
MLP, Pretrained	128	0.003908	0.003909
CNN	126	2.69e-21	0.006996
MLP	73	0.004286	0.008182
Univariate	98	0.004243	0.004606
LDA	64	0.005063	0.007880

The results in Table 6.3 show that using the pretrained models on our data does not yield a successful classifier. The models trained solely on the M0 data provide interesting results: the CNN has a high mean probability, but a very low median probability and median rank. This is to do with the CNN being ‘confident but wrong’; when the CNN predicts k^* correctly, it does so with high probability. However, it does not predict k^* correctly many times. The MLP on the other hand is much better at predicting the correct value, showing a median probability greater than uniform, and comparable to the univariate templating method. However, both models are outperformed by LDA in terms of median rank and median probability. We hypothesise that the model’s lack of generality is due to the fact that the ASCAD networks were trained on masked data (see Section 2.3.6.1), so the network was forced to learn the bivariate distribution of the mask and the real value.

6.3.3 Conclusion

Firstly, using pretrained networks is not always the most sensible approach, especially when there is a discrepancy between the previous problem and the new problem (in our case, possibly due to a mismatch in the ASCAD data being masked). Secondly, the Convolutional Neural Network was ‘confident but wrong’, which in most applications (e.g. image classification) is not always a bad thing. However, as we intend to use these networks to assist in a Belief Propagation Attack, this feature is severely detrimental, as we will show in the results. We prefer a network that provides consistent results, and the ASCAD MLP is a step in the right direction. However, it is by no means optimal, as it is outperformed by the Linear Discriminant Analysis classifier. We must consider re-tuning the network to maximise our own objective function with respect to our M0 data.

6.4 Hyperparameter Selection

The task of manually testing and selecting optimal hyperparameters is time consuming and arduous. There exist methods of speeding this process up; grid search and random search are tools that test a set of hyperparameters and select the best one automatically [39]. However, this work provides insight into how one would find the hyperparameters manually, and the effects each hyperparameter has in the context of power value classification.

This section looks at each hyperparameter (and modelling technique) used in the search to find the ‘best’ MLP for the M0 data. As we will eventually be using this model to classify all intermediates, but training on all intermediates during the hyperparameter selection phase would be too computationally expensive, we select our target to be the first output of the `SubBytes` operation s_1.

We used the ASCAD model as a starting point; once we experimented with a new hyperparameter and found a locally optimum value, we saved the value and used it in the subsequent experiments, progressively shaping the model into the ‘best’ MLP.

6.4.1 Window Size

The first hyperparameter we considered was the input length, known as the ‘window size’. This is the number of neurons (nodes) in the input layer. The ASCAD model used a window of 700 nodes, but (as far as we can tell from the paper) did not experiment with other window sizes.

The power value selection is defined in Equation 6.6, where T is the trace from which to extract the power values, t is the time point associated with the target intermediate, and w is the window of power values v to select.

$$\texttt{window}\,(T,t,w) = T[v_{t-\lceil \frac{w}{2} \rceil}, \ldots, v_{t+\lfloor \frac{w}{2} \rfloor}] \tag{6.6}$$

6.4.1.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_layers 6
```

```
-mlp_nodes 200 -epochs 200 -batch_size 200 -window [2, 10, ..., 900]
```

Window Size	**Mean Rank**	**Median Rank**
2	128.04	127
10	120.53	118
20	111.76	104
50	106.39	98
100	102.56	91
200	83.23	67
500	67.13	50.5
600	71.68	57
700	66.7	50
800	66.81	50
900	71.02	54

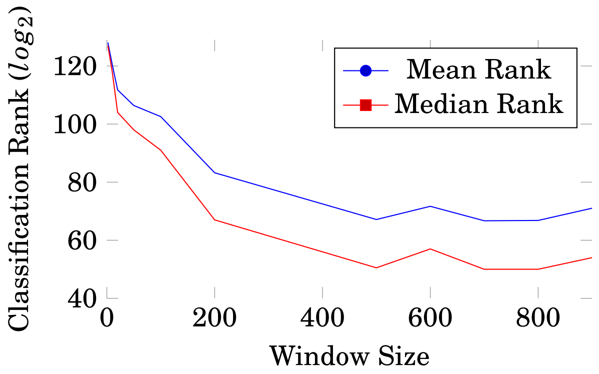


Figure 6.1: Table and Plot of Classification results tuning Window Size Hyperparameter

6.4.1.2 Observations

The best classification rank (mean and median) uses 700 neurons in the input layer. This is identical to the ASCAD value, which suggests that the model (as it stands) is unable to make use of additional power values, and the window of 700 perfectly encapsulates all necessary leakage information on s_1. This large window suggests that information leaks over a number of clock cycles, rather than leaking on a single clock cycle.

The univariate templating method has a median classification rank of 95, which means in its current state the Neural Network is outperforming the univariate method.

6.4.2 Number of Epochs

The learning algorithm works through the entire training dataset over a number of *epochs*. In a single epoch, each sample in the training set has had an opportunity to update the parameters within the network. Each epoch is made up of a number of *batches*, which we will be testing separately (see Section 6.4.5). An epoch with a single batch is called the 'gradient descent' learning algorithm. In general, classification success improves as we increase the number of training epochs (increasing the time taken) to a certain point, then we see diminishing returns. The ASCAD MLP uses 200 epochs for their 'best' model.

6.4.2.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_layers 6
-mlp_nodes 200 -window 700 -batch_size 200 -epochs [10, 20, ..., 7000]
```

Epochs	Mean Rank	Median Rank
10	109.27	100
20	86.84	72
50	69.81	53
100	56.33	40
150	55.78	40
200	51.69	37
250	55.53	40
300	48.11	33
400	46.12	31
500	42.79	28
750	39.59	25
1000	38.66	24
1200	35.33	22
1500	34.47	21
3000	30.79	18
4000	28.54	16
5000	27.79	15
6000	26.91	15
7000	27.62	16

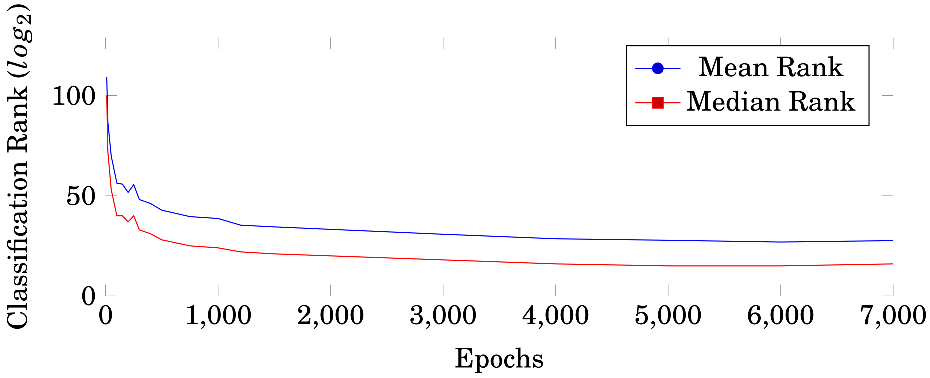


Figure 6.2: Table and Plot of Classification results tuning Number of Epochs

6.4.2.2 Observations

The classification rank continued to improve when we increased the number of epochs past the optimal value of 700 for the ASCAD data. It took 6,000 epochs to minimise the classification rank on the M0 data. With the current hyperparameters, it takes 3 seconds to perform one epoch over the training set. This equates to 5 hours training time (using 6,000 epochs).

One possibility why we found the optimal number of epochs to be much higher than the ASCAD value is that our data is slightly more complex; although we do not attack a masked implementation, the device on which our AES runs uses a 32 bit processor, yet the AES implementation is 8 bit. That means there are 24 bits unaccounted for, and we are unaware of what these bits do during computation, as they are independent of the AES implementation. As such, there is more for the network to learn, and by giving it more time in which to learn, it is able to refine its knowledge of the leakage model.

6.4.3 Hidden Layers

All layers in an MLP are fully connected; every node is connected to every other node in the subsequent layer. The universal approximation theorem states that “networks with two hidden layers and a suitable activation function can approximate any continuous function on a compact domain to any desired accuracy” [36, 37]. When dealing with more complex functions (as we do in our use case), we must experiment with the number of layers to find the optimal value. The ASCAD MLP uses 6 layers (4 hidden layers).

6.4.3.1 Results

We experiment not only changing the number of hidden layers, but with two different epoch values: 200 (as was found to be optimal for the ASCAD data) and 6,000 epochs (as was found to be optimal for our M0 data). The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
-window 700 -batch_size 200 -epochs [200, 6000] -mlp_layers [3,4,5,6]
```

	200 Epochs		6,000 Epochs	
Hidden Layers	Mean Rank	Median Rank	Mean Rank	Median Rank
1	65.92	49	32.61	19
2	60.73	44	31.05	18
3	67.62	51	32.43	19
4	65.8	49	32.75	19

6.4.3.2 Observations

The best result comes from using 2 hidden layers, resulting in a 4 layer MLP. In addition, we can clearly see that using 6,000 epochs gives us a much better classification rank than 200 epochs.

6.4.4 Augmentation I

Recent literature in the side channel community has shown that one can improve the classification performance of the Neural Networks by augmenting existing traces [10]. This is particularly useful when training data takes a long time to produce, as one can synthetically create new data from existing data. One widely used approach is to add Gaussian noise to existing traces, and adding these 'noisy' traces to the training dataset.

6.4.4.1 Results

We have 200,000 real traces; any more and we augment existing traces. The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
        -mlp_layers 4 -window 700 -batch_size 200 -epochs 6000
            -traces [200000, ..., 290000] -sd [0, 1, ..., 500]
```

Traces	**Mean Rank**	**Median Rank**
200,000	31.85	19
210,000	33.76	20
220,000	36.68	22
230,000	34.48	20
240,000	34.11	20
250,000	35.47	21
260,000	38.85	24
270,000	36.08	22
280,000	35.58	21
290,000	36.77	22

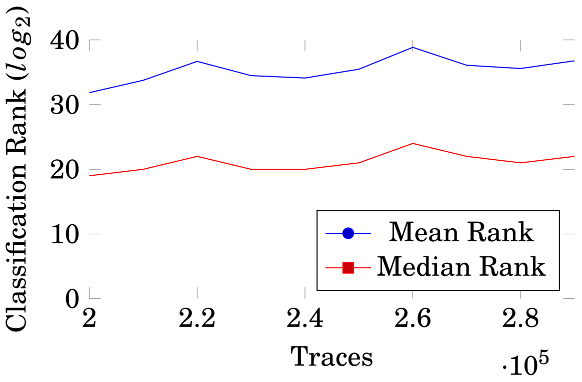


Figure 6.3: Table and Plot of Classification results tuning the Number of Augmented Traces (Gaussian Noise Standard Deviation = 100)

Standard Deviation	**Mean Rank**	**Median Rank**
0	36.99	22
1	34.82	21
5	37.43	23
10	37.18	22
20	35.62	21
50	36.55	22
100	36.96	22
200	38.08	23
500	41.76	27

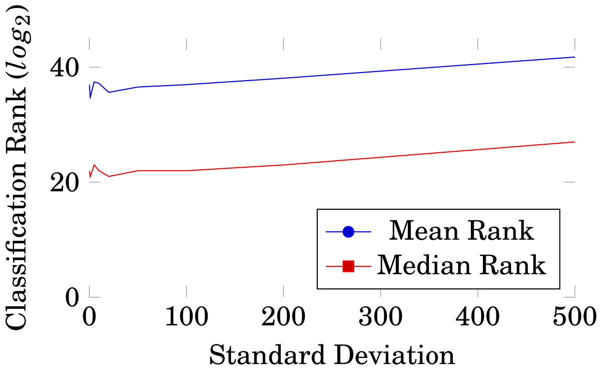


Figure 6.4: Table and Plot of Classification results tuning Data Augmentation Standard Deviation (10,000 additional augmented traces, totalling 210,000 training traces)

6.4.4.2 Observations

Unfortunately, using data augmentation in this way did not improve our classification results, with our best performance coming from not augmenting any traces at all. However, if we were to use augmentation, the best result uses a standard deviation of 1. This indicates that this method of augmentation does not excel on the M0 dataset.

6.4.5 Batch Size

As alluded to in Section 6.4.2, each epoch is made up of a number of *batches*. Altering the size of these batches has a large impact on the speed of training, and the quality of the trained network. A larger batch size allows for computational speedups through the use of parallelism in GPUs. However, the drawback is that the larger the batch size, the less generalisable the resulting network will be. We experiment with this hyperparameter on our M0 data.

6.4.5.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
                        -mlp_layers 4
-window 700 -epochs [200, 6000] -batch_size [10,...,2000]
```

Batch Size	Mean Rank	Median Rank
10	37.11	23
20	34.83	21
50	32.28	19
100	31.29	18
200	31.21	18
500	34.54	21
1000	37.05	22
2000	37.68	23

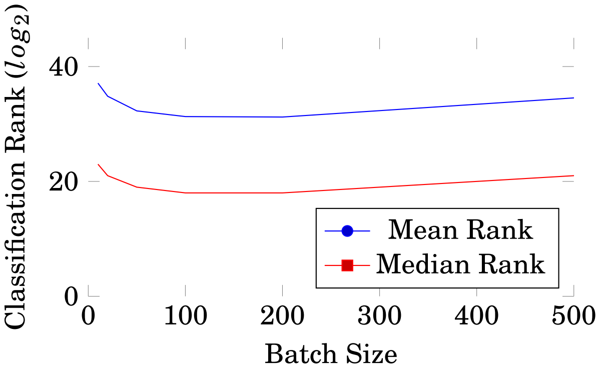


Figure 6.5: Table and Plot of Classification results tuning Batch Size

6.4.5.2 Observations

We find the best classification performance using a batch size of 200, which mirrors the value found for the ASCAD data.

6.4.6 Augmentation II

We revisit Data Augmentation by considering two additional augmentation techniques, used in recent literature [10].

1. Time Warping existing traces
   - This is done by 'shifting' an existing trace by a random amount (within a certain threshold)
2. Averaging two existing traces
   - Two traces (that have the same intermediate identity) are selected at random, and the mean of these is added to the augmented dataset
   - This method may not be effective as although we ensure the same identity value of our target intermediate, we don't ensure the same values of any other intermediate variables; e.g. the keys and plaintexts might not match, so some crucial information may be lost by averaging out

We compare these new techniques to our previous Data Augmentation experiments, which use Gaussian noise to generate new training data.

6.4.6.1 Results

In these results, instead of showing classification mean and median rank, we use the Neural Networks to perform a template attack. The results in Table 6.4 show the mean and median rank of the target subkey k_1 when attacking s_1 (the SubBytes output). The command used to generate these results was as follows, using red to indicate the modified parameters:

```
    python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
-mlp_layers 4 -window 700 -batch_size 200 -epochs 6000 -traces [200000, 290000]
              -aug [0 <Gaussian Noise>, 1 <Time Warp>, 2 <Averaging>]
```

	Template Attack Traces	
Augmentation Method	Mean Rank	Median Rank
None (200,000 Traces)	3.518	3.0
Adding Gaussian Noise (s.d. 100)	4.038	4.0
Time Warping (Max 10)	3.951	3.0
Averaging 2 Traces	4.045	4.0

Table 6.4: Table comparing Template Attack results using different methods of Data Augmentation (100,000 augmented traces, resulting in 300,000 training traces)

6.4.6.2 Observations

Echoing our previous results, we do not improve performance by using Data Augmentation. If we were to use augmentation, our experimental results suggest the use of the Time Warping technique, using a random shifting window threshold of 10 (the trace will only shift by a maximum of 10 samples each way).

6.4.7 Classifying Different Variables

Our experiments have led us to use the following hyperparameter values:

- 4 layer Multi-Layer Perceptron (2 hidden layers)
- 200 nodes per hidden layer
- Window of 700 power values
- 6,000 epochs with a batch size of 200
- 200,000 training traces (no data augmentation), 10,000 attack traces

The next step is to see how well the network model generalises for different intermediates. In an attack scenario, it would be too time and computationally expensive to find the optimal network structure for each intermediate (as we would need to repeat the above experiments 188 times). Our goal is to produce a network model that is able to generalise across all intermediates in our target cryptographic algorithm.

We selected three types of variables to test:

- the 16 key bytes k; the points of interest for these are either as the input to the `AddRoundKey` step, or in the first `KeyExpansion` step in AES
- the 16 output bytes of the `AddRoundKey` step t, used again as the input for the `SubBytes` step
- the 16 output bytes of the `SubBytes` step s, each used multiple times within the `MixColumns` step

These three variable types were chosen as they are the most 'important' nodes in the graph, shown in our experimental results in Section 4.7.5. For each intermediate, we train a separate network, using the network structure and hyperparameters listed above. We take the arithmetic mean and the median classification rank of 10,000 attack traces.

6.4.7.1 Results

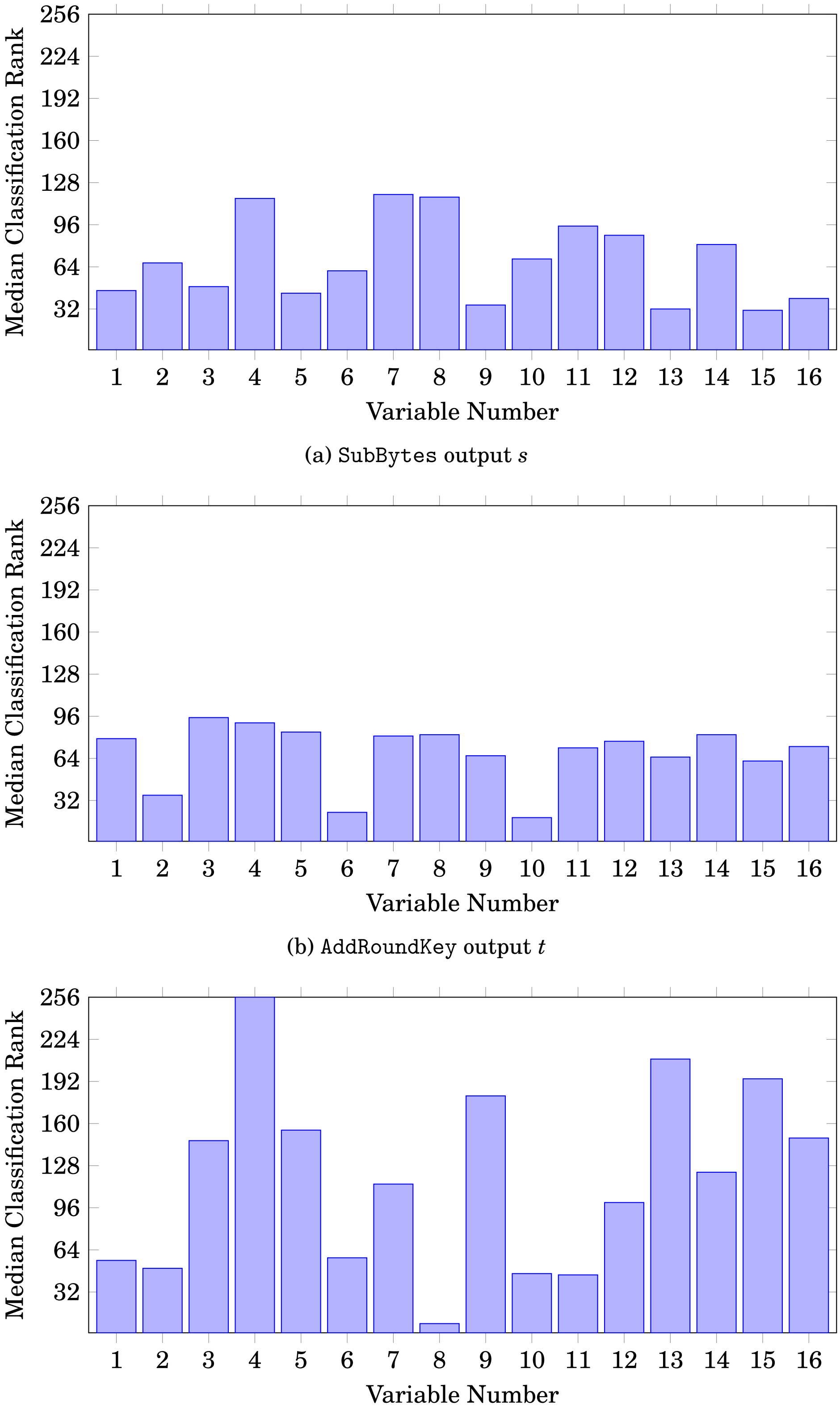
The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP
-mlp_nodes 200 -mlp_layers 4 -window 700 -epochs 6000
    -batch_size 200 -v [s001, ..., s016, t001, ...]
```

	Key Byte k		`AddRoundKey` t		`SubBytes` s	
Variable Number	**Mean Rank**	**Median Rank**	**Mean Rank**	**Median Rank**	**Mean Rank**	**Median Rank**
1	80.18	56	91.87	79	66.18	46
2	52.37	50	69.09	36	74.66	67
3	146.82	147	102.61	95	70.34	49
4	243.92	256	100.48	91	120.28	116
5	156.77	155	94.43	84	64.8	44
6	61.65	58	46.49	23	73.85	61
7	124.93	114	94.36	81	122.5	119
8	15.76	8	93.89	82	121.8	117
9	156.03	181	79.72	66	55.25	35
10	53.62	46	40.52	19	82.85	70
11	53.49	45	84.57	72	102.73	95
12	96.51	100	88.86	77	105.16	88
13	202.85	209	77.07	65	47	32
14	112.1	123	92.87	82	90.75	81
15	176.64	194	77.06	62	46.13	31
16	133.85	149	87.42	73	60.97	40

Table 6.5: Classification results for different intermediates

Figure 6.6: Table comparing the classification results of various intermediates



(a) `SubBytes` output *s*

(b) `AddRoundKey` output *t*

(c) Key Bytes *k*

Figure 6.7: Histograms comparing the classification results of various intermediates

6.4.7.2 Observations

Figure 6.7 shows the variance in the results, even across the same type of intermediate. Most notably, the classification ranks for the key bytes k fluctuate a lot; in the case of k_8, the network has a median classification rank of 8, which is a very good result. However, in the case of k_4, the network has a median classification rank of 256. This is because the network has a strong bias for a particular identity value; if the identity value of the inputted trace does not match this biased value, then the network fails to classify the trace.

These results are unusual; we would expect that the key bytes are more difficult to classify due to the lack of relevant leakage within the trace, but the results suggest that even for one type of intermediate (for instance, the `SubBytes` output s), the classification results vary in performance across the bytes, even though they use the same assembly instructions. This finding supports our hypothesis that we cannot produce a network that performance the 'best' over all intermediates.

6.4.7.3 Conclusions

The variance of median rank observed in the experimental results implies the networks are failing to generalise across all intermediates. From this we conclude that the current method of training the networks is insufficient; if a network trained to classify an intermediate is never able to classify the identity value correctly (e.g. k_4 in our experimental results), then we will never be able to achieve first order success when applying the networks to an inference-based attack.

6.4.8 Using the Networks in a Belief Propagation Attack

The results shown in Section 6.4.7 show that the Neural Networks generalise inconsistently across different intermediates. However, this is when each intermediate is classified separately. Using the Belief Propagation Algorithm, we can combine the information from multiple intermediates to recover information on the secret key.

Firstly, we run BPA using Neural Networks as the classification method (confirming the poor results echoed in the classification results). We do this by replacing the templating method with a Neural Network: when provided with a window of power values, the Neural Network produces a distribution of the possible values of some target node. This distribution is provided as the initial distribution for that target node in the factor graph, which is then propagated using the Belief Propagation Algorithm. We compare this to an attack that uses Neural Network classification when the Neural results have shown to be 'good' (classification rank less than 128), and supplies a uniform distribution to the 'bad' nodes (referred to as 'Ignore Bad' in Figure 6.8). We also show an attack where we use the 'best' classification method for that node (out of Univariate Templates, Linear Discriminant Analysis (window size 200), and the Neural Networks shown in Section 6.4.7).

6.4.8.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100 -raes 1
              --REAL [<none>, --LDA, --NN, --IGB, --BEST]
```

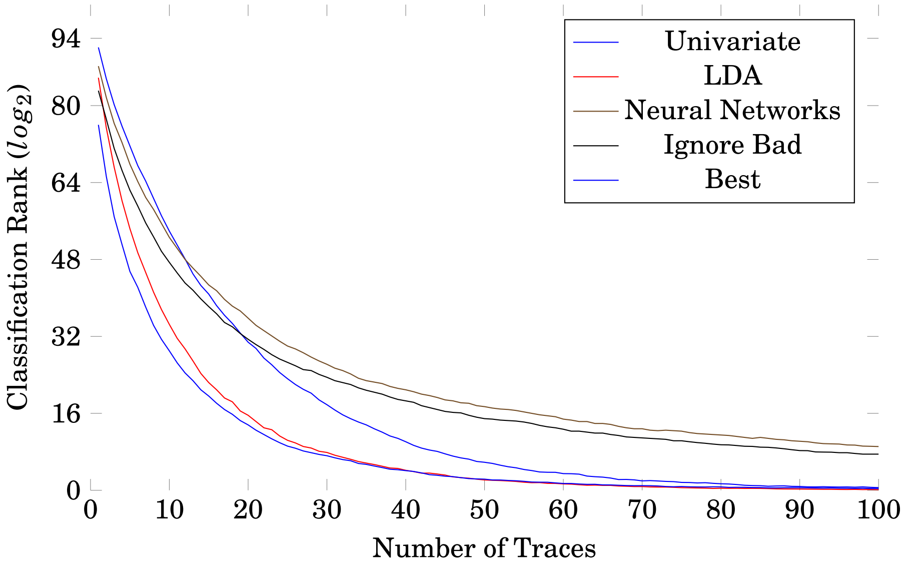


Figure 6.8: Plot comparing attack success using different templating methods

6.4.8.2 Observations

Using the Neural Networks as the sole classifier was not only the worst performing attack, but it failed to recover the correct key within 100 traces. This attack is only slightly improved by replacing the 'bad' networks with a uniform distribution. The best attack result came from combining the classification methods and selecting the 'best' classifier for each intermediate (lowest median classification rank).

6.4.9 Multi-Label Classification

Following from the results in Section 6.4.8, we must rethink our strategy for training the Neural Networks. One idea is to classify the *bits* of the identity byte value separately, rather than attempting to predict the byte value itself. This is known as *multi-label* classification. For example, in the case of three bits, the value 5 would have usually been one hot encoded to

$$[0,0,0,0,0,1,0,0]$$

whereas under the multi-label encoding it will be encoded as

$$[1,0,1]$$

to effectively be the binary representation of the value.

There are two main changes to the model that come as a consequence of multi-label classification:

1. The last layer activation function was softmax, now it must be sigmoid
   - Softmax produces a probability distribution over all possible values, but as we are predicting each bit independently, we have an output layer of nodes equal to the number of bits (in this case, 8)
2. The loss function categorical cross entropy must now be binary cross entropy
   - Each bit has two possible values: 0 or 1, and binary cross entropy reflects this

Following the results shown in Figure 6.7, we now include the *Common Prediction* percentage in our tables, shortened to ‘ComPred’. This value shows how often the network predicts the same value (chosen as the most common value for each network) during testing. This is because for some variables (e.g. k_4), the network predicts the same value throughout the majority of the testing dataset. We capture this information in this table, and we wish to use networks that have a low common prediction percentage.

6.4.9.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
-mlp_layers 4 -window 700 -batch_size 200 -epochs 6000 [<none>, --MULTI_LABEL]
```

		Single Label		Multi-Label	
Variable	Epochs	Median Rank	ComPred %	Median Rank	ComPred %
s_1	5	126	46.56	104	47.91
s_1	1000	82	14.8	92	17.13
k_4	6000	256	99.03	199	43.07

Table 6.6: Classification results comparing single label to multi-label encoding

6.4.9.2 Observations

When we train for a few epochs, multi-label approach has a better median rank. This implies it is quicker to learn about the target leakage function. However, after more training, the single label approach outperforms the multi-labelling.

Interestingly, the multi-label approach improves the k_4 classification by reducing the common probability percentage down from 99% to 43%, but the median classification rank is still above the uniform 128, so this improvement is not extremely beneficial.

6.4.10 Rank Loss Function

The next improvement idea came from questioning the use of categorical cross entropy as the loss function in the context of Side Channel Analysis. Cross entropy is defined in Equation 6.5. Due to the nature of the equation, it heavily penalises predictions that are confident but wrong. We do not want to penalise predictions that are 'wrong' (the highest predicted value is not the correct identity value), but instead encourage a high probability of the correct value relative to other predictions.

We now consider the 'Rank' loss function; identical to categorical cross entropy, but with the addition of the 'rank' of the correct value. By adding this value, we aim to minimise the rank of the correct identity value. As the rank is a value between 0 and 255, this will heavily outweigh the cross entropy value, but such balancing can be considered after an initial investigation.

6.4.10.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
-mlp_layers 4 -window 700 -batch_size 200 -epochs 6000 [<none>, --RANK_LOSS]
```

	Cross Entropy		Rank Loss	
Variable	Median Rank	ComPred %	Median Rank	ComPred %
s_1	46	5.58	51	6.89
k_4	256	93.64	219	77.69

Table 6.7: Classification results comparing Cross Entropy loss to Rank loss

6.4.10.2 Observations

We see similar results to the Multi-label approach; the performance classifying s_1 is worse using the Rank Loss function, but marginally improves classification for k_4 (but still below uniform).

In addition to the results shown in Table 6.7, we also experimented with *Hinge Loss* [72] (the most common loss function for support vector machines), but this did not suit our use case, and provided poor results.

6.4.11 Hamming Weight Classification

We have tried training the models to predict the identity function (either through single or multi-label) with little success. We now try predicting the Hamming Weight of the value. The following adjustments need to be made to accommodate this change:

1. The one hot encoding must be applied to the *Hamming Weight* of the value, rather than the identity value itself

2. The output layer must consist of 9 nodes, corresponding to the 9 possible Hamming Weights (0 to 8)

6.4.11.1 Results

The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/train_models.py --MLP -mlp_nodes 200
            -mlp_layers 4 -window 700 -batch_size 200
             -epochs 6000 [<none>, --HAMMING_WEIGHT]
```

	Identity Function			Hamming Weight		
Variable	Median Rank	Min Rank	ComPred %	Median Rank	Min Rank	ComPred %
s_1	55	1	6.65	126	1	40.72
k_4	256	1	94.97	154	28	87.31

Table 6.8: Classification results comparing Identity Function classification to Hamming Weight classification

6.4.11.2 Observations

Unfortunately, we see a similar result to the Multi-label method and the Rank Loss function: the classification results are much worse for the *SubBytes* output, but seem to improve upon the key bytes (though not below uniform). Of course, as we are predicting the Hamming Weight of the identity value, we will only achieve a rank of 1 if the network successfully predicts the value 0 or 9, as there exist multiple values that share the Hamming Weights 1 to 8.

6.5 Metric Re-evaluation

The hyperparameter results from Section 6.4 show poor generalisability over multiple intermediates. We experimented with various techniques to improve the classification rate, but to no avail. Now, we take a step back and re-evaluate whether our 'rank' metric is suitable in this context. This method was used during the training of the ASCAD model, so it was initially adopted as the main metric from which to select the optimal hyperparameters.

6.5.1 Rank as metric

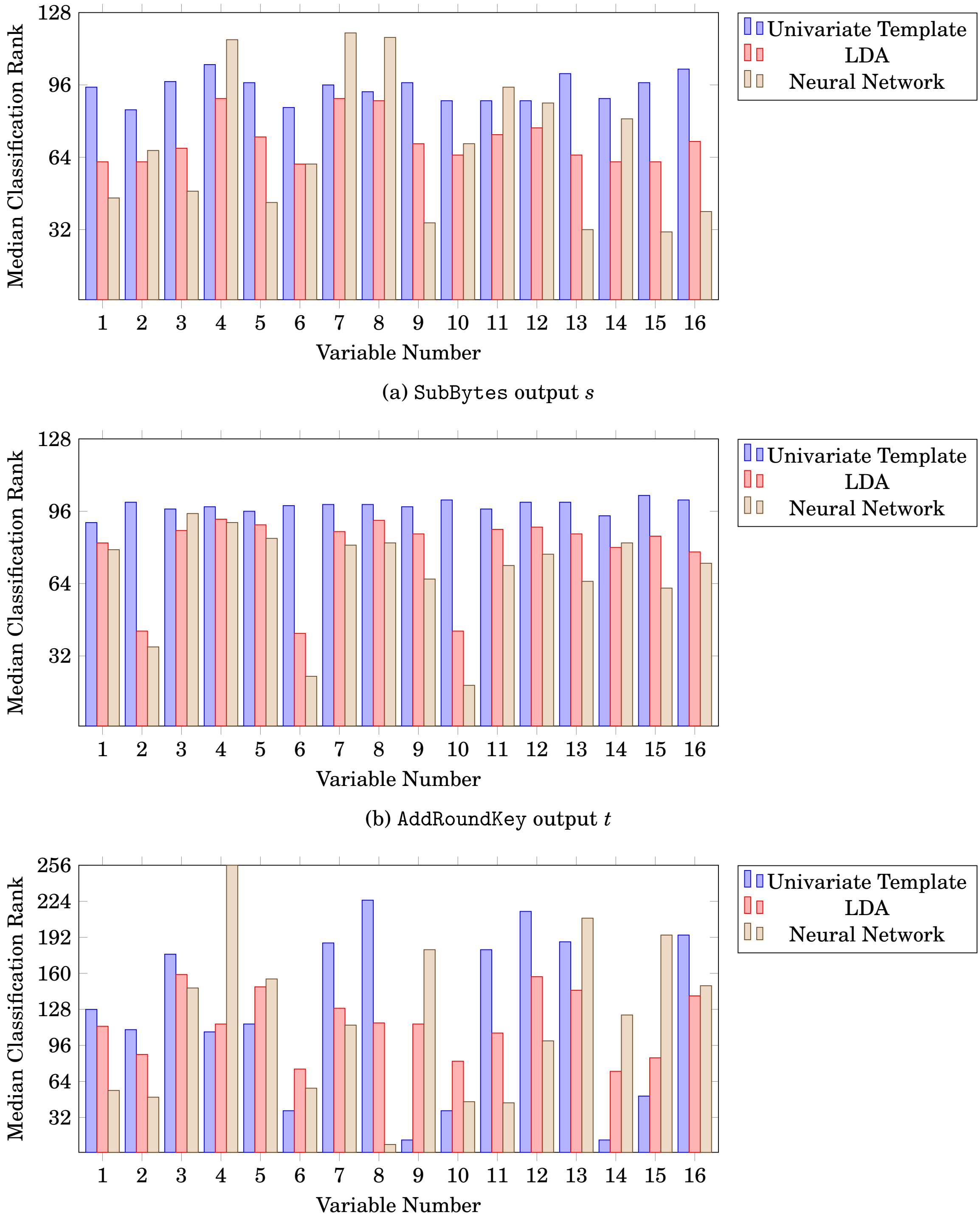
Figure 6.9a compares the median classification rank of three separate templating methods: the univariate templates, the LDA classifier, and the 'best' Neural Network found that maximises the median rank for s_1, the first `SubBytes` output. This is where we see something interesting: the performance of the classical profiling methods (univariate templating and Linear Discriminant Analysis) are also extremely variable. Similarly, Figure 6.9b targets the `AddRoundKey` output intermediate, which also has a fair amount of variance in the result. In this case, however, the Neural Networks consistently provide better success than the classical templating methods.

Finally, Figure 6.9c shows the classification results targeting the initial 16 key bytes. The leakage on the key bytes is often noisier and less reliable than those on the `SubBytes` outputs, and our results not only confirm this, but show the greatest variability yet. Again, this variance is also seen in the classical templating methods.

After observing these results, we conclude that perhaps this phenomenon is occurring not due to the *classifiers* behaving erratically, but due to *the way we measure the performance* (using the median rank). The rank (defined in Equation 6.1) is a useful measure that relates to how we evaluate attack outcomes in an attack scenario; however, in an attack scenario, we use multiple traces, which would produce stable ranks. The classification experiments we performed show the classification *per trace*, which means it is not unlikely that the classifiers will produce erroneous ranks. This is just one of the drawbacks of using the rank metric on per trace classification; another is that we are discarding information on the actual probability of the correct value, which would show how confident the network was at classifying the correct value. A metric that would capture this confidence would be using the *median probability*.

6.5.2 Probability as metric

The median probability metric is defined in Equation 6.3. As we are now using a different metric, we must re-tune all our hyperparameters with respect to the new metric. Upon doing so, the result was a completely different network structure than the one that maximised median rank. Table 6.9 shows the optimal hyperparameter values for the ASCAD MLP, the MLP using median rank as a metric, and our new MLP that uses median probability as a metric. Note that the output



(a) `SubBytes` output s

(b) `AddRoundKey` output t

(c) Key Bytes k

Figure 6.9: Three histograms showing the classification results of various intermediates using Median Rank as a performance metric

layer must use the Softmax activation function to ensure the network outputs a normalised probability distribution.

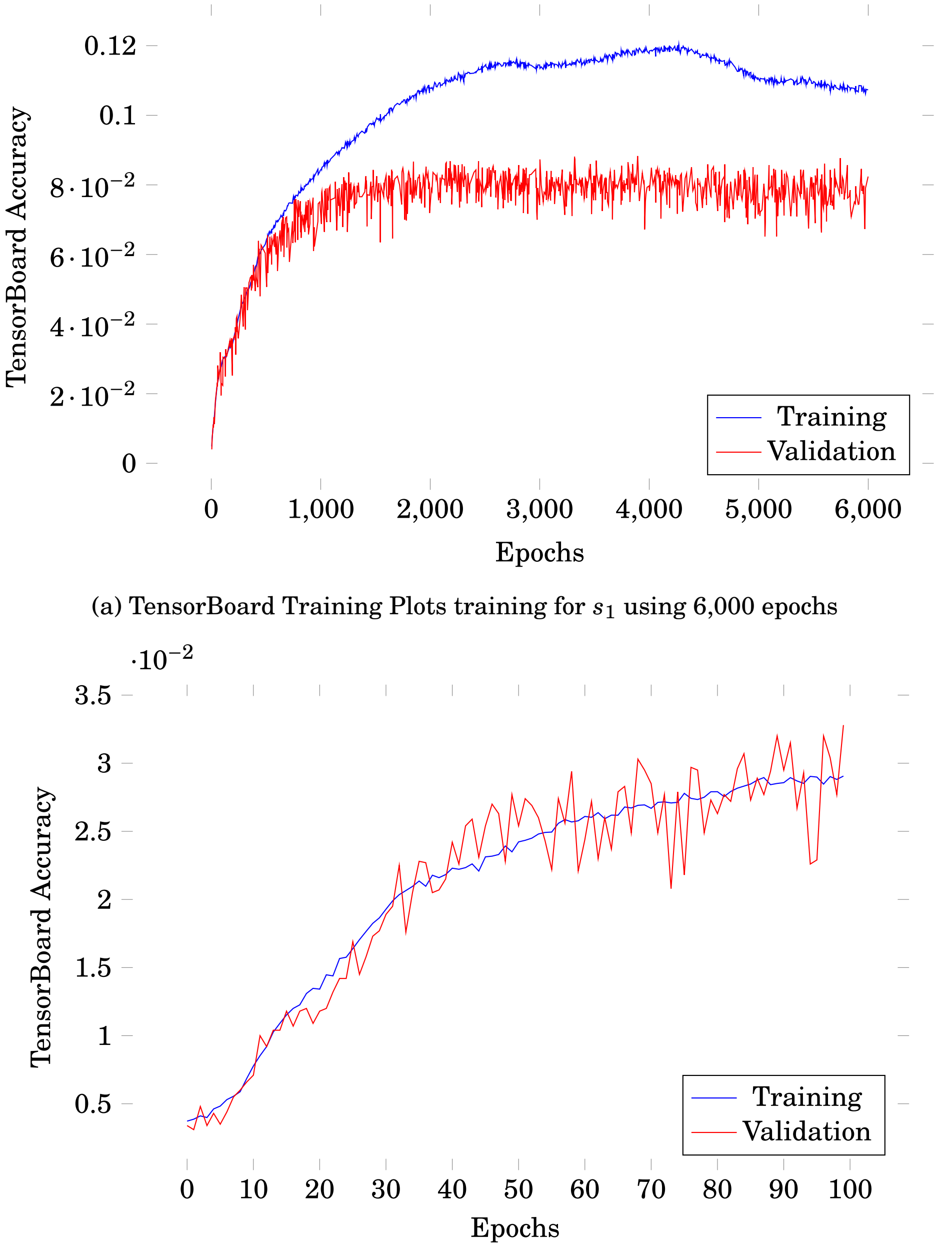
Parameter	ASCAD Network	Rank Network	Probability Network
Number of Hidden Layers	4	2	3
Number of Nodes in Hidden Layers	200	200	100
Activation Function	ReLU	ReLU	ReLU
Number of Epochs	200	6,000	100
Window Size	700	700	2,000
Batch Size	100	200	50
Learning Rate	10^{-5}	10^{-5}	10^{-5}
Optimiser	RMSProp	RMSProp	RMSProp

Table 6.9: Table comparing the locally optimal parameter values between various networks

There are some notable observations:

- Our new network has 3 hidden layers, in between ASCAD's 4 and the Rank network's 2, but the number of nodes per layer is decreased to 100
- The number of epochs has been dramatically reduced from the rank networks 6,000 (at 16 seconds per epoch makes 1 day to train) to 100 (27 minutes of training time)
- The window size has been dramatically increased from the previous 700 inputs to 2,000 units; this may be an indication of leakage present in this larger window that the probability network was able to harness (there is a buffer between the registers and the external memory in our target M0 device, which causes intermediate values to "hang around" for additional clock cycles)
- The optimal batch size dropped to 50, smaller than both the ASCAD and the Rank network

An interesting observation comes from looking at the TensorBoard accuracy plots taken during training. When we train the probability network using the values listed in Table 6.9, but this time using 6,000 epochs, we get the accuracy plot shown in Figure 6.10a. We can see from this plot that the network stops learning (the training accuracy peaks) at around 4,200 epochs, and the validation plot shows the model overfits after around 2,000 epochs. However, when we use our median probability metric to compare the results, the model using 100 epochs outperforms the model using any other number of epochs (6,000, 4,200, and 2,000). The training plot using 100 epochs is shown in Figure 6.10b, where the training and validation accuracy are still rising, showing there is still information 'to be learnt'. It is interesting that there is a mismatch between these plots and the actual effectiveness of the models (with respect to median probability); this discrepancy is most likely due to the accuracy plots using categorical cross entropy to calculate the training and validation accuracy.



(a) TensorBoard Training Plots training for s_1 using 6,000 epochs

(b) TensorBoard Training Plots training for s_1 using 100 epochs

Figure 6.10: TensorBoard Training Plots training for s_1 using different numbers of epochs; network parameters maximising the Median Probability metric

6.5.3 No Free Lunch

Now we have found a model structure and hyperparameters that maximise the median probability for the SubBytes output s_1, our next step is to compare this to the classical templating methods, and check to see how well the model generalises for other intermediates. Figure 6.11a is a histogram that compares the median probability classification of the probability based network to the classical templating methods (univariate templating and Linear Discriminant Analysis classifier) for the first 16 SubBytes outputs. At a glance, we can already see this is much more stable than the results shown in Figure 6.9a (the rank based network for the same intermediates), which gives us more confidence in the quality of the networks.

There exist some targets (e.g. s_2, s_9) where the Neural Networks outperform the classical classification methods; however, this is not always the case (e.g. s_8, s_{11}). This is also true for the AddRoundKey outputs and the key bytes, shown in Figure 6.11b and Figure 6.11c respectively; there exist intermediates in which the most successful classifier is not the Neural Networks, but one of the classical templating methods (e.g. t_3, k_{11}).

6.5.4 Conclusion

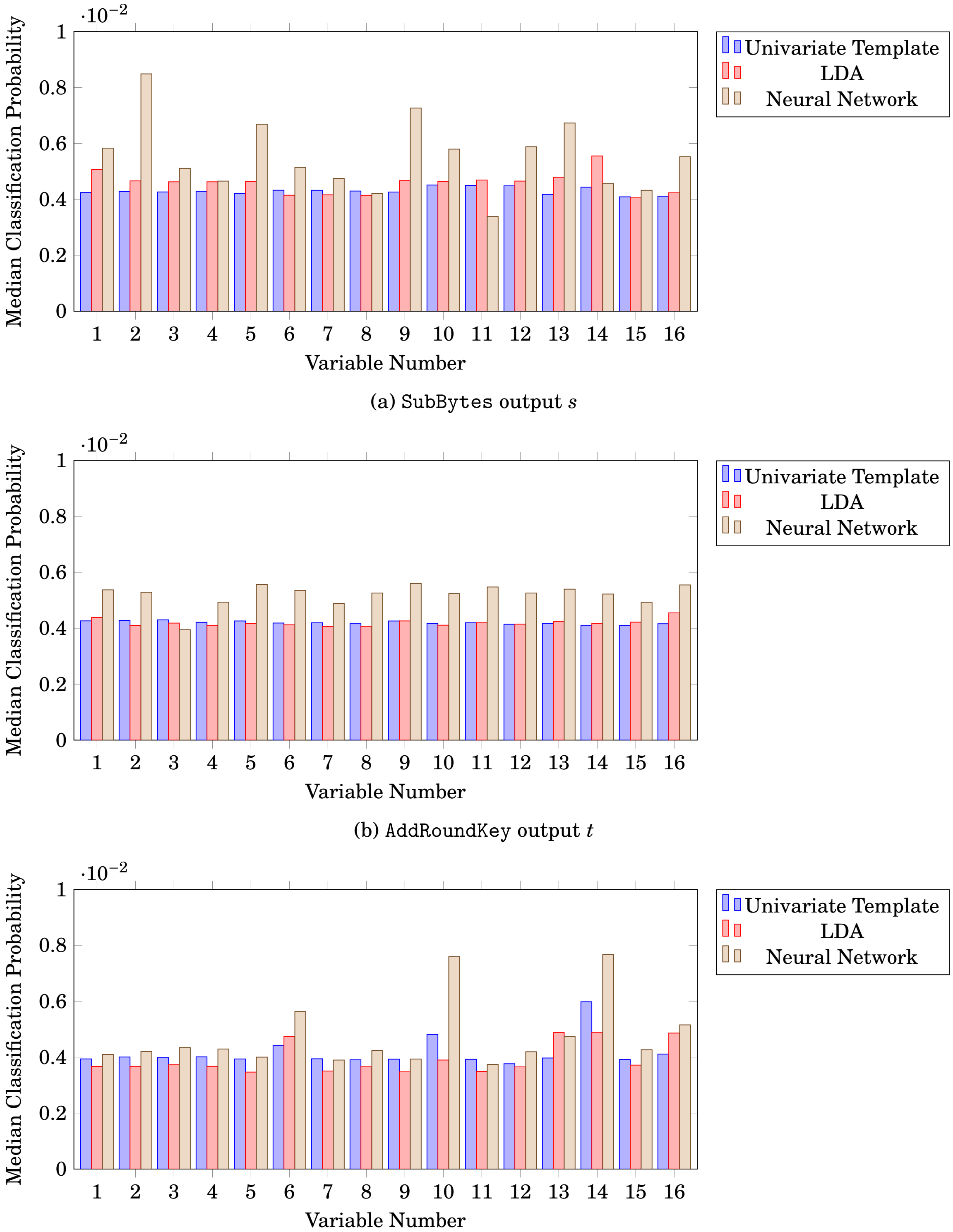
The stability of the probability based networks (compared to the rank based networks) give a more optimistic review of the performance of Neural Networks as a classifier. However, as seen in the results, they are not always the best classifier (compared to the classical methods). We believe this is a manifestation of the ‘no free lunch’ theorem; it is possible that there is no single classification method that outperforms all other learning approaches for all intermediates (or even for a specific type of intermediate, e.g. the first 16 SubBytes outputs). However, we can clearly see that the Neural Networks provide the best classification performance for the majority of intermediates, which implies they should perform well when used as the templating method in a profiled Side Channel Attack.

6.6 Belief Propagation with Neural Classifiers

In the previous section, we considered the classification performance of the Neural Networks (along with the classical templating methods) on a per trace basis. We now consider a full attack that combines leakage information from multiple traces. We start by solely attacking a single operation: the SubBytes step.

6.6.1 Attacking the SubBytes Step

When we use a classification tool on a window of leakage data (or single leakage value, in the case of Gaussian univariate templates), we produce a probability distribution for the likely values. By taking the product of these probability distributions for a number of traces (where the key is



(a) `SubBytes` output s

(b) `AddRoundKey` output t

(c) Key Bytes k

Figure 6.11: Histogram showing the classification results of various intermediates using Median Probability as a performance metric

fixed) followed by a normalisation phase, we get a probability distribution that has the combined information from all of the traces. If we know the plaintext and the key used for all the traces, we can permute this probability distribution to represent a distribution over the key bytes, rather than the SubBytes output, as shown in Equation 6.7.

$$k_i = \mathrm{SBOX}^{-1}(s_i) \oplus p_i \tag{6.7}$$

To actually implement the attack, we can make use of the Belief Propagation system built as described in Section 4.2. By ignoring all intermediates apart from the first sixteen SubBytes output bytes (and removing those nodes further away, resulting in reduced graph G_0), we can reproduce the single target attack. The nodes that are ignored will have a uniform distribution, which allows the probability distributions from the SubBytes output to propagate back to the key bytes without interference from any other distribution (they will only be permuted, as the XOR operation is with the plaintext byte, which is a one-hot encoded vector). The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 3 -t 100 -rep 100 -raes 0
                    --REAL [<none>, --LDA, --NN]
```

Figure 6.12a shows the mean final rank of the whole key by using different classifiers[3]. Also included is a non-profiled Hamming Weight based Correlation Power Analysis attack, an attack used when an adversary does not have access to a copy of the device, and thus cannot generate profiles for the target device's leakage (this was implemented separately, not using the Belief Propagation code). The results are intuitive; the profiled attacks outperform the non-profiled CPA, and the Neural Network assisted attack (the Neural Network here being the probability network) outperforms all other classifiers, by achieving first order success within 60 traces.

6.6.2 Combining Intermediate Leakages

By using the Belief Propagation Attack, we can combine leakage information not only from multiple traces, but from multiple leaking intermediates in each trace. For a description on how the Belief Propagation Algorithm works, see Section 2.5. Our attack implementation is similar to that in Section 6.6.1, but this time we do not ignore any intermediates, and we include all nodes from graph G_2. The command used to generate these results was as follows, using red to indicate the modified parameters:

```
python belief_propagation_attack/main.py -r 100 -t 100 -rep 100 -raes 2
                    --REAL [<none>, --LDA, --NN]
```

[3] We can use the mean rank here as we are combining information from multiple traces

Templating Method	Time Taken to generate 1 template	Memory Required for 1 Intermediate	Memory Required for all nodes in G_2 (188)
Gaussian Univariate Templates	45 seconds	2KB	376KB
Linear Discriminant Analysis Classifier	1 minute	3.2MB	601.6MB
Neural Network	27 minutes	4.6MB	864.8MB

Table 6.10: Time and Memory comparison of the different classification methods

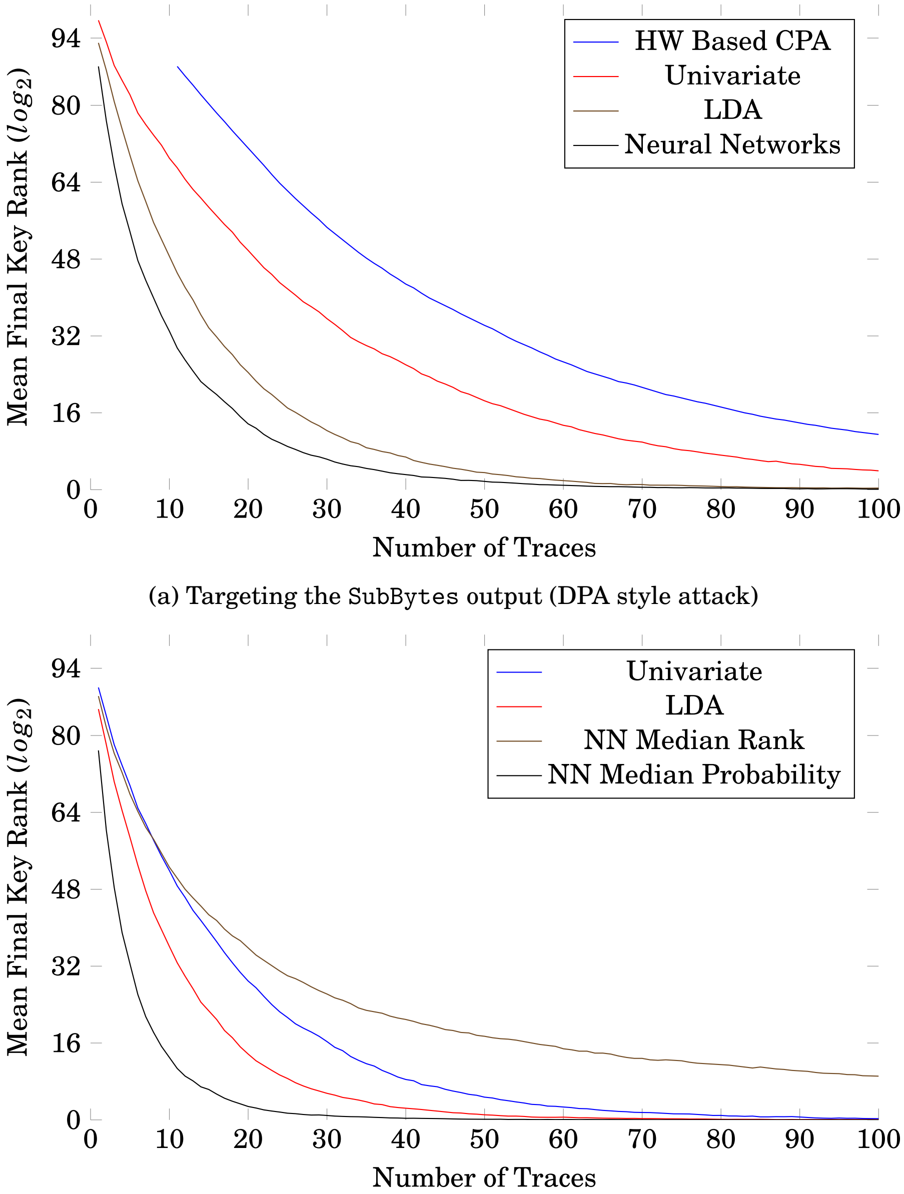
Figure 6.12b compares the attack results after running the Belief Propagation Attack using the three different classifiers: univariate templates, Linear Discriminant Analysis, and the Neural Networks (both the rank network and the probability network have been included for comparison). The rank based network performs very poorly compared to the other classification methods; this was foreshadowed by the variance of classification results over the different intermediates shown in Figure 6.9. The best attack performance came from the probability based network, which achieves first order success after 30 traces. It is able to bring the key space down to 2^{32} with less than 10 traces, which is easily enumerable on a standard PC, and therefore extremely successful as a classifier.

6.6.3 Conclusion

When the classification methods are compared in an attack scenario, the best results use Neural Networks as the classifier. However, this is in consideration of the *attack* phase, having already built all the necessary profiles (in this case, already produced the Neural Networks required). Table 6.10 compares the time and memory requirements for the different classifiers. The templates can be computed in parallel, so we only need to concern ourselves with the time taken to generate a single template. Neural Networks take 30 times longer than the LDA method, which is only marginally slower than generating the Gaussian Templates. However, in terms of memory, there is not a large difference between the LDA classifiers and the Neural Networks (although of course the univariate templates are exceedingly small, comprised of 256 * 2 32-bit floats).

An adversary that has access to a great deal of compute power should then opt to use the Neural Networks to aid in the classification phase, due to their performance over other classification methods. However, it should be noted that if time and compute power is not a factor, the adversary could afford to tune a separate model targeting every single required intermediate.

One question that merits further research is the appropriate use of the loss function in conjunction with the separate Median Rank / Probability metrics. During the training of the neural networks, the loss function is intended to be maximised / minimised, and the neural networks learn to classify data with respect to this loss function. It makes sense for this loss



(a) Targeting the `SubBytes` output (DPA style attack)

(b) Targeting all intermediates in G_2 using the Belief Propagation Attack

Figure 6.12: Plots showing results of different attacks targeting SubBytes and the whole G_2 graph

function to be identical to the intended use of the neural networks (in this case, in the context of the Belief Propagation Algorithm). However, early results indicated that by using categorical cross entropy as the sole metric (as seen in other work [15]), we achieve poor results in the Belief Propagation Algorithm. From this result stems our use of a separate metric (Median Probability / Rank), but a more efficient solution would be to create a loss function (compatible with keras / TensorFlow) that, when maximised / minimised, produces optimal results when used in conjunction with the Belief Propagation Algorithm.

Chapter 7

Concluding Remarks

This thesis sheds light on the worst-case physical security of cryptographic implementations by increasing our understanding of the most powerful class of side-channel adversaries. We improve both the *offline* and the *online* phase of an inference based attack on AES. In Chapter 4 we improved the *online* phase through the practical improvements of the Belief Propagation Algorithm. By studying the effect of Belief Propagation when applied to side channel analysis, we reduced the memory complexity by a magnitude whilst preserving the attack success, along with speeding up the runtime of the attack.

We explored the effect of convergence in the Belief Propagation Algorithm, and showed that by removing the cycles in the graph we can guarantee convergence. This reduces the runtime at the cost of a small amount of information. We also showed how we can connect multiple traces together in different ways, depending on the resources available to the attacker; if the attacker has a great deal of memory and compute power, they can combine all traces into a large graph, whereas if they have limited memory then the traces can be computed in parallel whilst maintaining a low memory requirement.

In Chapter 5 we described our target device along with our attack setup, and how an adversary could mount an attack against it. We started to consider improving the *offline* phase through the use of the multivariate Linear Discriminant Analysis classifier.

Finally, Chapter 6 covered our work with neural networks. We contributed our findings in the form of step-by-step experiments, manually tuning hyperparameters to create the 'best' network model structure to generalise across all our intermediates. We found that it was crucial to choose the right metric with which to judge network performance; we used two different metrics ('rank' and 'probability'), and showed direct comparisons between the network results. The probability based network outperformed the rank network substantially, and also outperformed the classical classification methods (univariate templating and Linear Discriminant Analysis). Our results followed the 'no free lunch' theorem, in that there was no single network that outperformed both classical methods for every single intermediate.

7.1 Assessment of Contributions

7.1.1 Belief Propagation

The paper titled *Soft Analytical Side-Channel Attacks* by Veyrat-Charvillon et al. [7] provided the first construction of a side channel attack using Belief Propagation. The contributions made in this thesis take the form of 'improvements' to this attack, by both modifying the graph structure representing AES, or by adding additional functionality to reduce the runtime and memory overhead of the attack. However, there are some clear distinctions between the attack setup used in this thesis and in the work done by Veyrat-Charvillon et al. The target device used by Veyrat-Charvillon et al. was an 8-bit Atmel ATMEGA644p microcontroller at a 20 MHz clock frequency. The target device used in this thesis was a 32-bit ARM Cortex-M0 at a 50 MHz clock frequency. This 24 bit overhead may have affected the noise in our results, as we were running the same 8-bit implementation of `AES FURIOUS` that Veyrat-Charvillon et al. run in [7].

The results shown in this thesis only used this one device, and only this specific implementation of AES. In order to truly make general statements, one could argue the experiments should be repeated on a separate device with a different implementation of AES.

One of our experiments analysed the impact of removing cycles in the factor graph representation, to guarantee convergence in the Belief Propagation algorithm (see Section 4.8.3). To remove the cycles, we studied the factor graph manually and selected nodes and edges to remove that would result in an acyclic graph. However, the nodes we chose were arbitrary; there are many other ways one could remove the cycles in the graph. We do not explore the other possibilities in this work, nor do we provide a method of generally choosing which edges to remove for any given factor graph. The method of node removal is left to the user.

7.1.2 Neural Networks

The neural network contributions made in this thesis are compared to those made in *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* by Prouff et al. [15]. However, these comparisons must consider the differences in attack setup (similar to the previous section). Most notably, the target AES implementations differ substantially; we use `AES FURIOUS`, but Prouff et al. use a masked AES implementation. This masking may have affected how the neural networks 'learnt' the leakage. When we attack the Sbox in our AES implementation, the neural network learns the univariate distribution of this leakage point. The networks developed by Prouff et al. must learn the bivariate distribution between the SubBytes output and the mask.

In addition to the paper, Prouff et al. provide code on GitHub [18], where users can run similar experiments to those shown in the paper. Reproducing the results proved to be a challenging task; it is unclear whether this was due to a difference in hardware or a problem with their code. This might be related to the fact that the CNN displayed poor results on the M0 data, although the MLP was able to classify the data successfully.

7.2 Future Work

This thesis studies the Belief Propagation attack when targeting AES, specifically the `AES FURIOUS` implementation. The contributions made in this thesis could be further extended to different algorithms, perhaps even asymmetric encryption algorithms (e.g. RSA). The Belief Propagation algorithm is completely dependent on the structure of the graph, and this would change significantly if using a different cryptographic algorithm.

In addition, Belief Propagation is just one of the many well-known message passing algorithms. The *"Divide and Concur"* algorithm [73] is another, which aims to satisfy constraints with continuous variables. It is worth looking into these other methods as an alternative to Belief Propagation because (at the time of writing) these have not been studied in great detail when applied to Side Channel Analysis.

As mentioned in the analysis, we selected nodes and edges arbitrarily to remove in order to achieve an acyclic graph. Further study could look into this in more depth; is there a method that will consistently select the edges and/or nodes to remove to achieve an acyclic graph that minimises the information loss? This would be useful and would encourage the use of guaranteed converging factor graphs.

Finally, the work presented by Prouff et al. in *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database* [15] shows that the Multi-Layer Perceptron outperforms the Convolutional Neural Network when classifying power leakage when there is no jitter present. However, when jitter is introduced, the CNNs outperform the MLPs. Further study could extend the contributions made in this work to consider clock jitter, and other implementations of side channel countermeasures. There exist several methods of dealing with clock jitter; one can either mitigate the jitter through statistical preprocessing (dynamic time warping) or one can train a network (CNN) to adapt to the jitter (or even do a mixture of both). Further study could aim to answer the question: what is the best way to deal with jitter in an inference-based attack?

APPENDIX A

APPENDIX

A.1 Belief Propagation Attack

Table A.1: Table listing all argument flags available when running `main.py` from the command line. The lower case flags require an additional input (e.g. `-t <number>` runs the attack with a set number of traces), whereas the upper case flags are booleans that toggle the default result when provided to `main.py`.

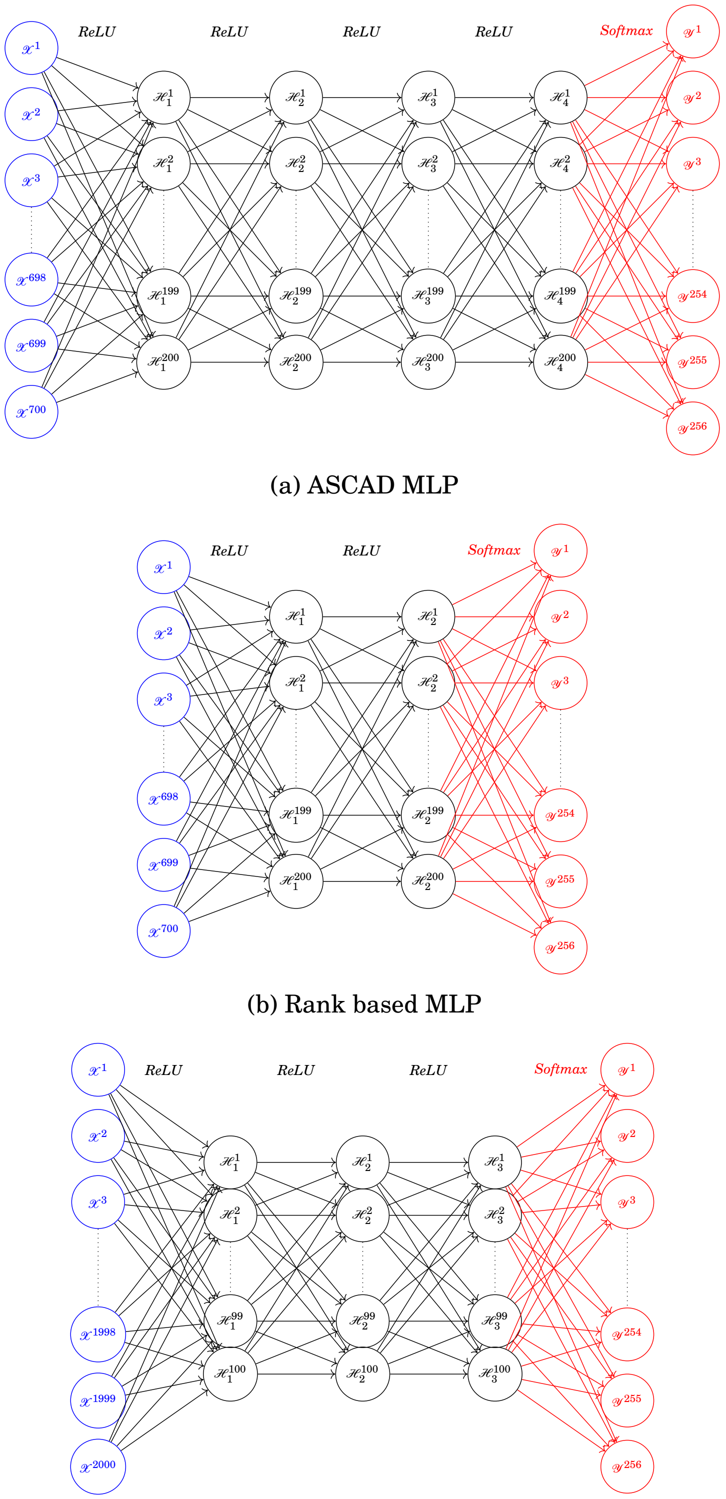
Flag(s)	Description	Default Value
	BP Algorithm Parameters	
rep	Number of Repetitions to Average	1
r	Number of Iterations in BP	5
t	Number of Traces	1
	BP Algorithm Tweaks	
epsilon	Threshold for breaking early	0.0001
epsilon_s	How many successive epsilon round to break early	10
IGT	Toggles Ignore Ground Truths	TRUE
BERR, BIF	Break if failed (check through Plaintext)	FALSE
BFND	Break when correct value found	FALSE
BPAT	Break when patterns matched	FALSE
	Factor Graph Structure	
lo	For Distance between nodes, leave out node	[]
rm	Removes target variable	[]

Flag(s)	Description	Default Value
raes, rounds_of_aes	Number of Rounds of AES	2
ING, IND, IFG	Toggles Independent Graph Off	TRUE
KS	Toggles Key Scheduling On	FALSE
LFG	Toggles Large Factor Graph On	FALSE
REMOVE_CYCLE, RM_C, ACYCLIC	Removes cycle in MixColumns step	FALSE
SQG, SEQ, SFG	Toggles Sequential Graph On	FALSE
ARM, ARM_AES	Use ARM AES Implementation instead of AES Furious	FALSE
	Input Data	
bl	Badly leaks on target variable	[]
blt	Badly leaks on target traces	[]
nn	No noise on target variable	[]
nl	Doesn't leak on target variable	[]
blsnrexp	snr Exponent for the bad leakage, s.t. snr = 2**SNR_exp	-7
ct, cthresh	Threshold for refusing bad point of interest detected nodes	None
j, jitter	Clock Jitter to use on real traces	None
k	Chosen Key as Hex String	None
seed	Seed for extra randomisation	0
snrexp	snr Exponent, s.t. snr = 2**SNR_exp	5
thresh	Threshold for refusing bad leakage	None
tp, trace_range	Window of Power Values over Time point	1
KEYAVG, KAVG, KPAVG	Toggles Key Power Value Averaging Off	TRUE
LOTF	Toggles Leakage on the Fly off	TRUE
LOCAL_LEAKAGE	Toggles Local Leakage Off, if cannot compute on the fly	TRUE
RANDOM_REAL	Uses Random Trace Subsets for Real Trace Experiments	TRUE
UNPROFILED, NEW	If Real Traces, only attack unprofiled traces	TRUE
ELMO	Toggles ELMO Power Model On	FALSE
HW	Toggles HW Power Model On	FALSE
IGB, IGNORE_BAD_TEMPLATES	Toggles Ignore Bad Templates	FALSE
NLKS	Doesn't leak on Key Schedule	FALSE
NO_NOISE	No Noise in Simulation	FALSE
READ_PLAINTEXTS	Reads Plaintexts from File	FALSE
REAL_TRACES, REAL	Attacks a Real Trace	FALSE
UPDATE_KEY, UKID	Updates Key Initial Distributions	FALSE
USE_BEST, B, BEST	Uses Best Template for Real Traces, out of Univariate, LDA, and NN	FALSE

Flag(s)	Description	Default Value
USE_LDA, LDA	Uses LDA for Real Traces	FALSE
USE_NN, NN	Uses Neural Network for Real Traces	FALSE
	Misc	
fix	Fix Variable node to get Marginal Distance to Key Bytes	None
	Printing and Storing Results	
ALL_CSV	Writes all data to csv file after each repeat	FALSE
CONVERGENCE_CSV	Writes convergence data to csv file after each trace and repeat	FALSE
CONVERGENCE_TEST	Prints out Convergence Statistics	FALSE
DUMP_RESULT, DATA_DUMP	Dumps Result in output/data_dump.txt	FALSE
FULL_ROUND_CSV	Writes full key rank round data to csv file	FALSE
MARTIN, MARTIN_RANK	Adds Martin Rank to final Rank	FALSE
NO_PRINT	Only prints out Average Rank	FALSE
ONLY_END, END_ONLY	Only prints out End Result	FALSE
ONLY_FINAL, FINAL_ONLY	Only prints out Final Rank	FALSE
ONLY_RESULT, RESULT_ONLY, RESULTS_ONLY	Only prints out Result	FALSE
PLOT	Plot Final Key Ranks	FALSE
PRINT_DICT	Prints Dictionary of Values from Leakage Simulation	FALSE
EVERY_TRACE	Prints out Key Rank after each trace	FALSE
FKD	Prints Final Key Distribution	FALSE
RANK_CSV	Writes key rank data to csv file	FALSE
REPEAT_CSV	Writes key distribution data to csv file after each repeat	FALSE
ONLY_REP, REPEAT_ONLY	Only prints out End of each Repeat Result	FALSE
ROUND_CSV	Writes key rank round data to csv file	FALSE
SAVE_FIRST_DISTS, SFD	Saves the Distributions of the first Key and Plaintexts bytes (Test Purposes Only)	FALSE
TRACE_CSV	Store Trace Values as csv	FALSE
TRACE_NPY	Store Trace Values as npy	FALSE
WRITE_CSV, CSV	Writes key distribution to csv file	FALSE
WRITE_NPY, NPY	Writes key distribution to npy file	FALSE
	Testing and Debugging	
TEST_NAME	Name for Test (if testing)	Standard
CL, CHECK_LEAKAGE	Checks Initial Leakage	FALSE

Flag(s)	Description	Default Value
CURRENT_TEST	Alters values to match current test statistic	FALSE
RANDOM_KEY	Uses Random Key (Test Purposes Only)	FALSE
TEST_KEY, NEW_KEY	Uses Different Key (Test Purposes Only)	FALSE

A.2 Neural Networks



(a) ASCAD MLP

(b) Rank based MLP

(c) Probability based MLP

Figure A.1: MLP architectures

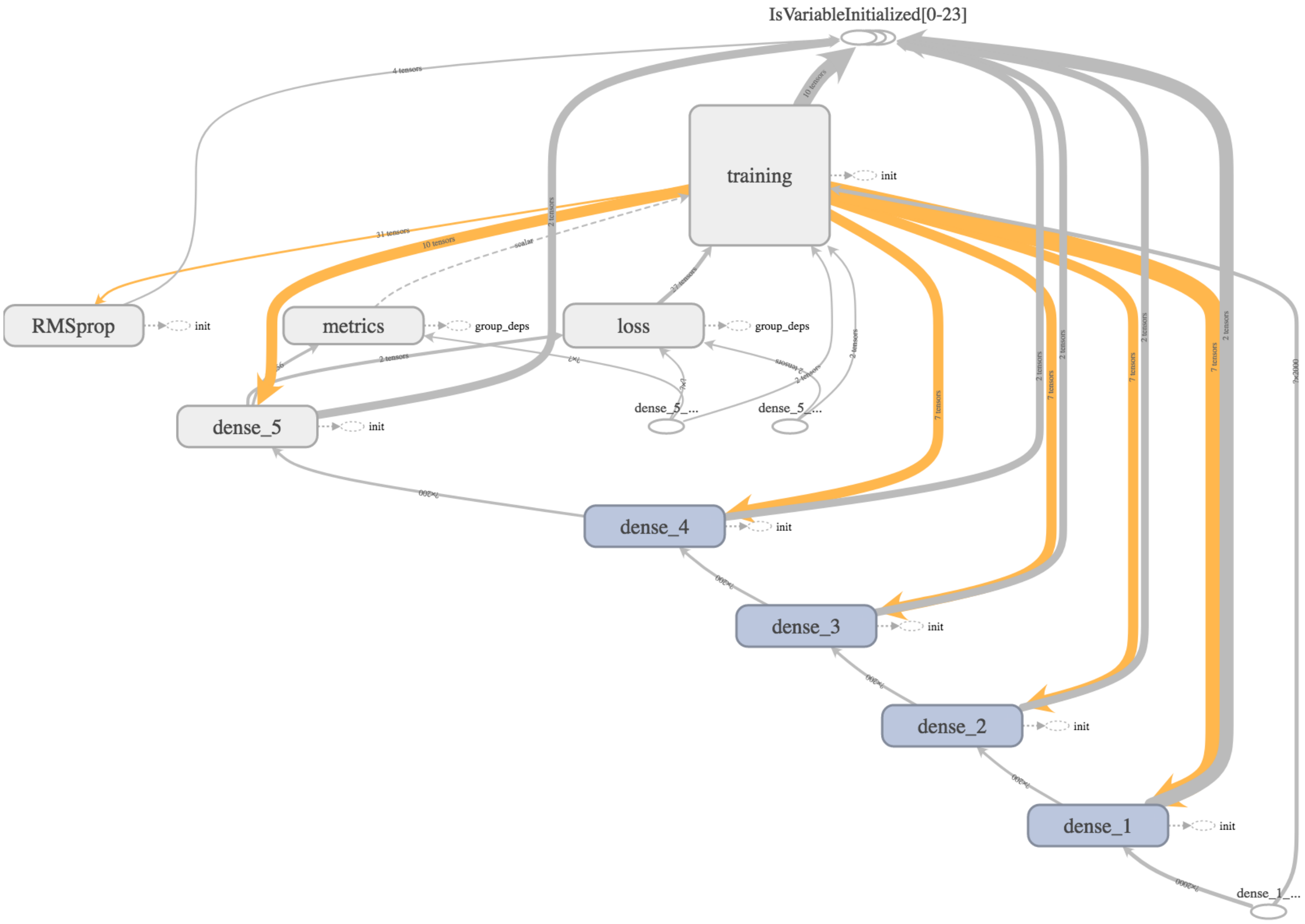


Figure A.2: Graphical Representation of the Probability Network, as generated by TensorBoard

Bibliography

[1] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, “Introduction to differential power analysis,” *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr 2011. [Online]. Available: https://doi.org/10.1007/s13389-011-0006-y

[2] J. Green, A. Roy, and E. Oswald, “A Systematic Study of the Impact of Graphical Models on Inference-Based Attacks on AES,” in *Smart Card Research and Advanced Applications*, B. Bilgin and J.-B. Fischer, Eds. Cham: Springer International Publishing, 2019, pp. 18–34.

[3] T. Halevi and N. Saxena, “Keyboard acoustic side channel attacks: exploring realistic and security-sensitive scenarios,” *International Journal of Information Security*, vol. 14, pp. 1–14, 09 2014.

[4] D. Yucebas and H. Yuksel, “Power analysis based side-channel attack on visible light communication,” *Physical Communication*, vol. 31, pp. 196 – 202, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1874490717304913

[5] P. C. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *CRYPTO’99*, ser. LNCS, M. J. Wiener, Ed., vol. 1666. Springer, Heidelberg, Aug. 1999, pp. 388–397.

[6] L. Mather, E. Oswald, and C. Whitnall, “Multi-target DPA attacks: Pushing DPA beyond the limits of a desktop computer,” Cryptology ePrint Archive, Report 2014/365, 2014, http://eprint.iacr.org/2014/365.

[7] N. Veyrat-Charvillon, B. Gérard, and F.-X. Standaert, “Soft analytical side-channel attacks,” in *ASIACRYPT 2014, Part I*, ser. LNCS, P. Sarkar and T. Iwata, Eds., vol. 8873. Springer, Heidelberg, Dec. 2014, pp. 282–296.

[8] M. Manoj krishna, M. Neelima, H. Mane, and V. Matcha, “Image classification using deep learning,” *International Journal of Engineering and Technology*, vol. 7, p. 614, 03 2018.

[9] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell, “Understanding data augmentation for classification: when to warp?” *ArXiv e-prints*, Sep. 2016.

[10] E. Cagli, C. Dumas, and E. Prouff, “Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures,” in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 45–68.

[11] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic, “Make Some Noise. Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis,” pp. 148–179, May 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8292

[12] Z. Martinasek, J. Hajny, and L. Malina, “Optimization of Power Analysis Using Neural Network,” in *Smart Card Research and Advanced Applications*, A. Francillon and P. Rohatgi, Eds. Cham: Springer International Publishing, 2014, pp. 94–107.

[13] Z. Martinasek, L. Malina, and K. Trasy, *Profiling Power Analysis Attack Based on Multi-layer Perceptron Network*. Cham: Springer International Publishing, 2015, pp. 317–339. [Online]. Available: https://doi.org/10.1007/978-3-319-15765-8_18

[14] Z. Martinasek, P. Dzurenda, and L. Malina, “Profiling power analysis attack based on MLP in DPA contest v4.2,” 06 2016, pp. 223–226.

[15] E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Dumas, “Study of deep learning techniques for side-channel analysis and introduction to ASCAD database,” Cryptology ePrint Archive, Report 2018/053, 2018, https://eprint.iacr.org/2018/053.

[16] J.-C. Sibel, S. Reynal, and D. Declercq, “Evidence of chaos in the Belief Propagation for LDPC codes,” 06 2012.

[17] D. McCann, C. Whitnall, and E. Oswald, “ELMO: Emulating Leaks for the ARM Cortex-M0 without Access to a Side Channel Lab,” *IACR Cryptology ePrint Archive*, vol. 2016, p. 517, 2016.

[18] ANSSI, “ASCAD Database,” 2018. [Online]. Available: https://github.com/ANSSI-FR/ASCAD

[19] J. Green, E. Oswald, and A. Roy, “A Systematic Study of the Impact of Graphical Models on Inference-Based Attacks on AES,” Cryptology ePrint Archive, Report 2018/671, 2018, https://eprint.iacr.org/2018/671.

[20] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin, Heidelberg: Springer-Verlag, 2002.

[21] B. Poettering. (2003) AVRAES: The AES block cipher on AVR controllers. [Online]. Available: http://point-at-infinity.org/avraes/

[22] P. C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” in *Advances in Cryptology — CRYPTO ’96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[23] B. G. Y. S. Fred de Beer, Marc Witteman, “Practical electro-magnetic analysis,” *Non-Invasive Attack Testing Workshop NIAT-2011*, 2011.

[24] P. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” in *Advances in Cryptology — CRYPTO’ 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[25] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[26] E. Oswald and K. Schramm, “An efficient masking scheme for AES software implementations,” in *WISA 05*, ser. LNCS, J. Song, T. Kwon, and M. Yung, Eds., vol. 3786. Springer, Heidelberg, Aug. 2006, pp. 292–305.

[27] J. Balasch, S. Faust, B. Gierlichs, and I. Verbauwhede, “Theory and practice of a leakage resilient masking scheme,” in *ASIACRYPT 2012*, ser. LNCS, X. Wang and K. Sako, Eds., vol. 7658. Springer, Heidelberg, Dec. 2012, pp. 758–775.

[28] S. Ahn and D. Choi, “An improved masking scheme for S-box software implementations,” in *WISA 15*, ser. LNCS, H. Kim and D. Choi, Eds., vol. 9503. Springer, Heidelberg, Aug. 2016, pp. 200–212.

[29] P. Karpman and D. S. Roche, “New instantiations of the CRYPTO 2017 masking schemes,” Cryptology ePrint Archive, Report 2018/492, 2018, https://eprint.iacr.org/2018/492.

[30] J. G. J. van Woudenberg, M. F. Witteman, and B. Bakker, “Improving Differential Power Analysis by Elastic Alignment,” in *Topics in Cryptology – CT-RSA 2011*, A. Kiayias, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 104–119.

[31] D. McCann, E. Oswald, and C. Whitnall, “Towards Practical Tools for Side Channel Aware Software Engineering: ‘grey box’ Modelling for Instruction Leakages,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann

[32] J. Pearl and T. Verma, “The Logic of representing Dependencies by Directed Graphs,” Cognitive Systems Laboratory, Los Angeles, CA, 1987.

[33] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[34] D. J. C. MacKay, *Information theory, inference, and learning algorithms*. Cambridge University Press, 2003.

[35] T. Tieleman and G. Hinton, “Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude,” COURSERA: Neural Networks for Machine Learning, 2012.

[36] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals and Systems*, vol. 2, 1989.

[37] K. Hornik, M. B. Stinchcombe, and H. White, “Multi-layer feedforward networks are universal approximators,” 1988.

[38] M. Telgarsky, “Benefits of depth in neural networks,” *CoRR*, vol. abs/1602.04485, 2016. [Online]. Available: http://arxiv.org/abs/1602.04485

[39] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 281–305, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2503308.2188395

[40] Google. (2018) TensorFlow: Machine Learning Framework. [Online]. Available: https://www.tensorflow.org/

[41] Google. (2018) TensorBoard: Visualizing Learning. [Online]. Available: https://www.tensorflow.org/guide/summaries_and_tensorboard

[42] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, “A testing methodology for side channel resistance,” 2011.

[43] Z. Lu and K.-H. Yuan, *Welch’s t test*, 01 2010, pp. 1620–1623.

[44] “ISO/IEC 17825:2016: Testing methods for the mitigation of non-invasive attack classes against cryptographic modules.” [Online]. Available: https://www.iso.org/standard/60612.html

[45] K. Chatzikokolakis, T. Chothia, and A. Guha, “Statistical Measurement of Information Leakage,” in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 390–404.

[46] T. Chothia and A. Guha, “A Statistical Test for Information Leaks Using Continuous Mutual Information,” in *2011 IEEE 24th Computer Security Foundations Symposium*, June 2011, pp. 177–190.

[47] L. Mather, E. Oswald, J. Bandenburg, and M. Wójcik, "Does my device leak information? an a priori statistical power analysis of leakage detection tests," in *Advances in Cryptology - ASIACRYPT 2013*, K. Sako and P. Sarkar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 486–505.

[48] F. Durvaux and F.-X. Standaert, "From improved leakage detection to the detection of points of interests in leakage traces," Cryptology ePrint Archive, Report 2015/536, 2015, https://eprint.iacr.org/2015/536.

[49] "The Common Criteria for Information Technology Security Evaluation." [Online]. Available: https://www.commoncriteriaportal.org/

[50] "EMVCo Homepage." [Online]. Available: https://www.emvco.com/

[51] C. Whitnall and E. Oswald, "A Cautionary Note Regarding the Usage of Leakage Detection Tests in Security Evaluation," Cryptology ePrint Archive, Report 2019/703, 2019, https://eprint.iacr.org/2019/703.

[52] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *CHES 2002*, ser. LNCS, B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, Eds., vol. 2523. Springer, Heidelberg, Aug. 2003, pp. 13–28.

[53] V. Grosso and F.-X. Standaert, "ASCA, SASCA and DPA with enumeration: Which one beats the other and when?" in *ASIACRYPT 2015, Part II*, ser. LNCS, T. Iwata and J. H. Cheon, Eds., vol. 9453. Springer, Heidelberg, Nov. / Dec. 2015, pp. 291–312.

[54] Y. Oren, M. Renauld, F.-X. Standaert, and A. Wool, "Algebraic Side-Channel Attacks Beyond the Hamming Weight Leakage Model," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 140–154.

[55] F. Durvaux, M. Renauld, F.-X. Standaert, L. van Oldeneel tot Oldenzeel, and N. Veyrat-Charvillon, "Efficient removal of random delays from embedded software implementations using hidden markov models," in *Smart Card Research and Advanced Applications*, S. Mangard, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 123–140.

[56] S. Song, K. Chen, and Y. Zhang, "Overview of side channel cipher analysis based on deep learning," *Journal of Physics: Conference Series*, vol. 1213, p. 022013, jun 2019. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F1213%2F2%2F022013

[57] "DPA contest," 2019. [Online]. Available: http://www.dpacontest.org/home/

[58] P. C. Team, "Python 2.7: A dynamic, open source programming language," 2015. [Online]. Available: https://www.python.org/

[59] S. Behnel, R. Bradshaw, L. Dalcín, M. Florisson, V. Makarov, and D. Sverre Seljebotn, "Cython C-Extensions for Python," 2018. [Online]. Available: https://cython.org/

[60] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 467–475. [Online]. Available: http://dl.acm.org/citation.cfm?id=2073796.2073849

[61] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Generalized belief propagation," in *Advances in neural information processing systems*, 2001, pp. 689–695.

[62] D. Page, "SCALE: Side-Channel Attack Lab. Exercises." [Online]. Available: http://www.github.com/danpage/scale

[63] H. Nyquist, "Certain Topics in Telegraph Transmission Theory," *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, April 1928.

[64] N. I. of Advanced Industrial Science and Technology. (2003) The original SASEBO FPGA board. [Online]. Available: http://point-at-infinity.org/avraes/

[65] RISCURE, "Inspector Side Channel Analysis Security." [Online]. Available: https://www.riscure.com/security-tools/inspector-sca/

[66] N. Homma and M. Medwed, Eds., *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 9514. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-31271-2

[67] W. Zheng, L. Zhao, and C. Zou, "An efficient algorithm to solve the small sample size problem for lda," *Pattern Recognition*, vol. 37, no. 5, pp. 1077 – 1079, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320303003261

[68] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *Trans. Evol. Comp*, vol. 1, no. 1, pp. 67–82, Apr. 1997. [Online]. Available: https://doi.org/10.1109/4235.585893

[69] D. Wolpert, "The supervised learning no-free-lunch theorems," 01 2001.

[70] "Keras: The Python Deep Learning library," 2019. [Online]. Available: https://keras.io/

[71] W. Schindler, K. Lemke, and C. Paar, "A Stochastic Model for Differential Side Channel Cryptanalysis," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 30–46.

[72] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri, "Are loss functions all the same?" *Neural Comput.*, vol. 16, no. 5, pp. 1063–1076, May 2004. [Online]. Available: http://dx.doi.org/10.1162/089976604773135104

[73] S. Gravel and V. Elser, "Divide and concur: A general approach to constraint satisfaction," *Physical Review E*, vol. 78, no. 3, p. 036706, 2008.