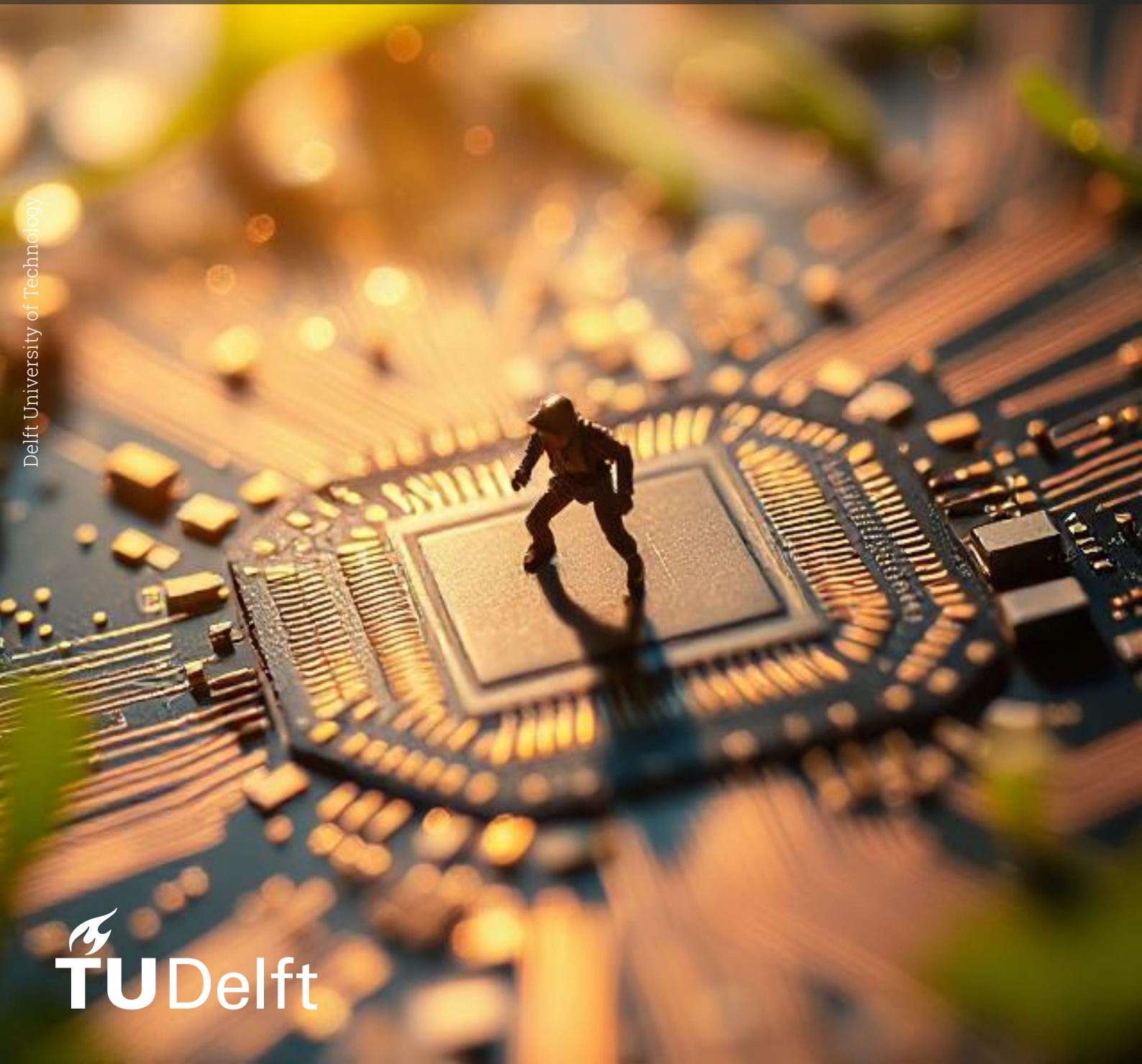
Deep Learning and Side-Channel Analysis

A Language Model-Inspired framework

João Martinho



Delft University of Technology

TU Delft

Deep Learning and Side-Channel Analysis

A Language Model-Inspired framework

by

João Martinho

Daily Supervisor:	Dr. Stjepan Picek
TU Delft supervisor:	Dr. Jakob Söhl
Committee member:	Dr. Alexander Heinlein
Project Duration:	November, 2024 - August, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science

TU Delft

Summary

As the world becomes increasingly digital, cybersecurity—particularly cryptography—has become a defining concern of this century. Beyond designing robust algorithms, it is vital to evaluate the resilience of devices to adversaries who exploit various aspects of algorithm execution. Side-channel analysis targets physical leakages, such as power consumption and electromagnetic emissions, to extract secret information. State-of-the-art research identifies machine learning attacks using Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) as the most effective. Language Models have achieved success across diverse domains, some unrelated to language. This thesis investigates their applicability to side-channel analysis and compares their performance with current state-of-the-art methods. Sane or Silly, a language model - inspired framework, is introduced and used to attack the ASCAD datasets. Results demonstrate that this approach can successfully retrieve the key in both ASCADf and ASCADv using only one trace, regardless of whether the secret masks are known during profiling. Desynchronization hindered but did not fully prevent successful attacks. These findings highlight the potential of language models as powerful tools for side-channel analysis.

Nomenclature

Abbreviations and acronyms used throughout this thesis are presented here to avoid confusion

Abbreviations

Abbreviation	Definition
AES	Advanced Encryption Standard
BO	Bayesian Optimization
CNN	Convolutional Neural Network
DL	Deep Learning
GE	Guessing Entropy
HD	Hamming distance (leakage model)
HW	Hamming Weight (leakage model)
ID	Identity (leakage model)
LLM	Large Language Model
ME	Metadata Embedding
ML	Machine Learning
MLP	Multi-Layer Perceptron
NOPOI	Not-Optimized Points of Interest
NTMP	Not-Trained Mask Predictor
OPOI	Optimized Points of Interest
PTMP	Pre-Trained Mask Predictor
RPOI	Refined Points of Interest
SCA	Side-Channel Analysis
SNR	Signal-to-Noise Ratio
SoS	Sane or Silly

List of Figures

List of Tables

Contents

1

Introduction

Information security and privacy are among the most important concerns of the current century. With the constant growth in the amount of data shared daily and the ever-increasing computing power, there is a rising demand for secure protocols to protect and transfer information. Cryptography is the field that investigates the most secure methods of sharing data, ensuring that it remains inaccessible to potential eavesdroppers. While cryptography—viewed from an information theory perspective—is of utmost importance and mathematically rich, it often simplifies real-world scenarios by assuming that an attacker has access only to the encrypted data. In reality, attackers frequently gain access to various leakages—such as power consumption and electromagnetic emanations—and combine this information with the intercepted ciphertext to compromise protocols that are theoretically secure.

Side-channel analysis (SCA) is a pivotal field within cryptography that studies attacks targeting the physical implementation of security systems. Traditionally, SCA has relied on classical techniques—such as statistical methods and hand-crafted heuristics—to interpret side-channel leakages including power consumption, electromagnetic emissions, and timing variations [17] [41] [18]. However, with significant advancements in computational power and the emergence of sophisticated machine learning techniques, the state of the art in SCA (as well as in many other fields) has notably transitioned toward the application of deep learning methodologies [30] [2].

Another notable trend in the machine learning community is the rise of language models, which have achieved unprecedented success in processing sequential data and uncovering complex patterns within unstructured text. Models with similar architectures have also achieved success in various domains - including decision-making [23], music generation [22], protein folding [24], and time series analysis [29] - sparking interest in studying the effectiveness of these approaches for side-channel analysis. In this thesis, one of the first explorations into leveraging a language model-like architecture for SCA is presented. The goal is to investigate which architectures exhibit the best performance in SCA while comparing language model–based approaches with the current state-of-the-art techniques.

This thesis is structured as follows: Section 2 presents a comprehensive background, introducing all the essential concepts required to understand both SCA and deep learning methodologies. Section 3 provides a review of the most relevant literature, summarizing key contributions and positioning this work within the broader research landscape. In Section 4, the methodology employed for designing and training the architecture is outlined, followed by Section 5 which details the experimental setup, where most details of the experiments are described - the more cumbersome details are left for the appendix. Section 6 presents the experimental results obtained from applying the model to a dataset, while Section 7 discusses these findings and their implications for the field. Finally, Section 8 concludes the thesis by summarizing the contributions, and a brief personal note is given on Section 9.

To promote transparency and facilitate future research, the entire code will be made publicly available. This thesis, therefore, not only endeavors to advance the practice of side-channel analysis through novel deep learning techniques but also seeks to bridge the gap between traditional SCA approaches and modern machine learning paradigms.

2

Background

2.1. Machine Learning

Machine learning is a research field centered on designing algorithms that, by learning from data, can autonomously discover general patterns through experience on a subset of the data. These algorithms harness statistical techniques to infer the underlying structure of data, enabling tasks such as classification, regression, and clustering without needing explicit programming [3].

Machine Learning tasks can roughly be divided into 2 mains categories: Supervised and Unsupervised Learning. In supervised learning, the data $X = \{X_1, ..., X_n\}, X_i \in \mathcal{X}$ is accompanied by labels $Y = \{Y_1, ..., Y_n\}, Y_i \in \mathcal{Y}$, and the task that is intended to be learned is to approximate the underlying function $f : \mathcal{X} \to \mathcal{Y}$ such that $f(X_i) = Y_i$. Furthermore, supervised learning tasks can be separated into classification tasks - where the labels are discrete - and regression tasks - where the labels are continuous. In unsupervised learning, no labels are provided, and the tasks consists of finding some underlying patterns in X. [3]

An important notion is that the goal is not to have a model that learned all the information about the data it was given. Instead, it is to teach the model all the underlying patterns and structure on the data, such that not only it learned about the data it has seen, but hopefully, it can infer facts about unseen data. A model that learned information about the data it has seen but struggles to generalize is said to be overfitting. To evaluate this, data X is usually split into training data, which will be fed to the model, and validation data, unseen data on which the model will be evaluated [3].

Over the last years, Machine Learning frameworks have been showing outstanding performance in various tasks, often replacing human made functions and statistical models. On the one hand, this replacing allows for creation of high performing frameworks that due to its complexity would not have been created otherwise, but on the other hand, replaces explainable and intuitive models with black-box ones.

2.2. Deep Learning

Deep learning is a specific set of techniques contained within machine learning that encompasses models whose structure is a Neural Network, structures consisting of several neurons, structures inspired by biological neurons, assembled in various ways [3]. This framework is used for supervised learning tasks.

Each neuron N is a simple $\mathbb{R}^n \to \mathbb{R}$ function whose parameters are a set of weights $w = \{w_1, ..., w_n\}$, a bias b, and a non-linear function σ. Its usual definition is the one given on equation (2.1).

$$N_{w,b,\sigma}(x_1, ..., x_n) = \sigma(\sum_{i=1}^{n} w_i \times x_i + b) \tag{2.1}$$

Neural networks themselves are structured as layers of these interconnected neurons - referred to as the input layer, hidden layers, and the output layer, as illustrated by Figure 2.1, where each circle corresponds to one neuron, and the whole diagram corresponds to a (very simple and shallow) neural network. The inputs of each neuron ($x_1, ..., x_n$) are a subset of the input data for the input layer, and a subset of the outputs of the neurons of the previous layer, for the other layers.

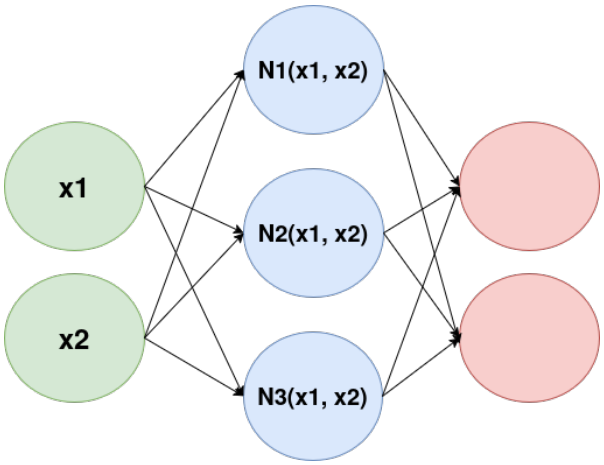


Figure 2.1: Simplified Neural Network. In green, the input layer, in blue one hidden layer and in red the output layer. In this case, there are 5 sets of weights, biases and activation functions (one in each neuron, except the ones in the input layer)

With this, neural networks are a family of parametrized functions. These parameters are the weights and the biases of all neurons. A supervised learning task, consists of searching for a set of parameters that results in a function that approximates $f\mathcal{X} \to \mathcal{Y}$ such that $f(X_i) = Y_i$. To turn this search problem into an optimization one, the performance of a model is measured by a loss function, a function $\mathcal{L} : \mathcal{W} \times \mathcal{B} \to \mathbb{R}$ which measures the distance between the model's predictions and data labels. The training process is therefore finding $(W, B) \in \mathcal{W} \times \mathcal{B}$ that minimize $\mathcal{L}(W, B)$ [3].

Within this framework, different architectures can arise, resulting from a different number of neurons assembled in a different way. Architectures are usually categorized in the way neurons are fed to the following layer, the most widely used being Convolutional Neural Networks (CNNs) and Multilayer Perceptrons(MLPs).

While the term parameter is usually employed to refer to the trainable weights and biases of the set of neurons, the term hyperparameters is often used to refer to aspects of the neural network's architecture of training process.

2.2.1. Activation Functions

Each neuron is equipped with a non-linear function σ. As the other operations executed by a neuron are linear, the concatenation of these operations without any other function would be also linear, resulting in a model that could only hope to correctly approximate linear functions, which in most cases would not be useful. Instead, a non-linear function σ, referred to in the ML literature as an activation function, is applied to the neuron's linear combination of inputs. Usually the outputs of a layer before applying an activation function are referred to as logits. Many different activation functions are experimented with in the literature, as tuning it is a difference maker in a model's performance. Here are some of the most popular choices:

Rectified Linear Unit (ReLU):

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2.2}$$

Scaled Exponential Linear Unit (SELU):

$$\text{SELU}(x) = \lambda \begin{cases} x, & \text{if } x > 0, \\ \alpha e^x - \alpha, & \text{if } x \leq 0, \end{cases} \tag{2.3}$$

where $\lambda \approx 1.0507$ and $\alpha \approx 1.67326$. These constants are chosen to ensure self-normalizing properties in neural networks.

Cosine Similarity:

$$\text{cosine similarity}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \sqrt{\sum_{i=1}^{n} v_i^2}}. \tag{2.4}$$

Cosine similarity is a measure of similarity between two vectors $\mathbf{u}$ and $\mathbf{v}$. In practice, as it is often preferred to work with values between 0 and 1, cosine similarity is usually linearly scaled to be between 0 and 1. This is true for this work.

Softmax Function:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, 2, \ldots, K. \tag{2.5}$$

Applied to a vector $\mathbf{z} = (z_1, z_2, \ldots, z_K)$ of real numbers. Each component z_i is exponentiated and then normalized by the sum of the exponentials of all components, ensuring that the output values form a probability distribution. Similarly to cosine similarity, the output is bounded, which usually is a desired property in output layers.

2.2.2. Loss Functions

As mentioned before, loss functions measure the performance of a model. They are functions $\mathcal{L} : \mathcal{W} \times \mathcal{B} \to \mathbb{R}^+$ that will be minimized during training. As the classification and regression tasks require different loss functions, some examples of loss functions are presented for both purposes:

Classification Loss Functions:

Cross-Entropy Loss: For multi-class classification, given a true one-hot encoded label vector $\mathbf{y} = (y_1, y_2, \ldots, y_C)$ and a predicted probability vector $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_C)$, the cross-entropy loss is defined as:

$$\mathcal{L}_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i). \tag{2.6}$$

Hinge Loss: Often used in Support Vector Machines for binary classification, with true labels $y \in \{-1, +1\}$ and prediction score $f(x)$, the hinge loss is given by:

$$\mathcal{L}_{\text{hinge}}(y, f(x)) = \max\left(0, 1 - y \cdot f(x)\right). \tag{2.7}$$

Regression Loss Functions

Mean Squared Error (MSE): For regression tasks with true values y_i and predictions $\hat{y}_i$ over N samples, the MSE is defined as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \hat{y}_i\right)^2. \tag{2.8}$$

Mean Absolute Error (MAE): The MAE is similar to the MSE, but takes the average of the absolute values instead of the squared values:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|\,. \tag{2.9}$$

Apart from these classical loss functions, many more loss functions arise in the literature, as there are specific problems that call for specific loss functions. Any non-trivial distance function in $\mathbb{R}$ is, in principle, a feasible loss function. Furthermore, it is possible to incorporate in the loss function the values of y_i and $\hat{y}_i$ separately, in addition to their difference, to indicate that errors for particular values of y_i or $\hat{y}_i$ are more severe than for others (Weighted Loss Functions).

2.2.3. Training Process and Optimizers

A training session of a model consists in feeding the model data, evaluating the loss function, and adjusting the model's parameters accordingly. The data is not fed all at once, but instead, in batches, allowing the model to adjust its parameters slowly. A bigger batch size allows batches which are more representative of the entire dataset, while a smaller batch size makes training more stochastic, with more variability from batch to batch, making the model face more and more diverse challenges, which can be good [16].

In machine learning, an optimizer is an algorithm that adjusts the parameters of a model to minimize a loss function. In general, it updates weights and biases by computing the gradient of the loss function and updating the parameters in the opposite direction (direction of steepest decline in loss function) They are parameterized by a parameter μ called learning rate, which controls how big the changes in parameters are from batch to batch.

The data is usually fed to the model more than once - the number of times each data point is fed to the model is usually referred to as epoch.

Stochastic Gradient Descent (SGD)

Updates the model parameters in the opposite direction of the gradient of the loss function. The update rule for a parameter θ is given by:

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta\mathcal{L}(\theta_t), \tag{2.10}$$

where η is the learning rate and $\nabla_\theta\mathcal{L}(\theta_t)$ is the gradient of the loss function $\mathcal{L}$ with respect to θ at time step t [10].

SGD with Momentum

Introduces a speed term s_t that accumulates the gradient's direction to speed up convergence:

$$\begin{aligned} s_{t+1} &= \gamma s_t + \eta\nabla_\theta\mathcal{L}(\theta_t), \\ \theta_{t+1} &= \theta_t - s_{t+1} \end{aligned} \tag{2.11}$$

where s_t is the speed, γ is the momentum coefficient (typically between 0 and 1), and η is the learning rate [10]. Introducing the concept of speed helps SGD incorporate past gradients, enabling it to dampen oscillations, making it more robust when traversing complex or noisy loss surfaces [33].

AdaGrad (Adaptive Gradient Algorithm)

Adapts the learning rate for each parameter individually by scaling it inversely proportional to the square root of the cumulative sum of squared gradients.

$$\begin{aligned} G_t &= G_{t-1} + (\nabla_\theta\mathcal{L}(\theta_t))^2, \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon}\nabla_\theta\mathcal{L}(\theta_t) \end{aligned} \tag{2.12}$$

where G_t is the sum of the squares of the gradients up to time step t, and ϵ is a small constant to avoid division by zero. While AdaGrad performs well on sparse data, its aggressively decreasing learning rate can cause premature convergence [10].

RMSProp

Improves on AdaGrad by using an exponentially decaying average $E[g^2]$ of squared gradients:

$$\begin{aligned} E[g^2]_t &= \rho E[g^2]_{t-1} + (1-\rho)(\nabla_\theta \mathcal{L}(\theta_t))^2, \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\theta \mathcal{L}(\theta_t) \end{aligned} \tag{2.13}$$

where ρ is the decay rate (typically around 0.9) and ϵ is a small constant for numerical stability [10].

Adam (Adaptive Moment Estimation)

Combines momentum and RMSProp by maintaining moving averages of both gradients and squared gradients:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t), \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned} \tag{2.14}$$

where β_1 and β_2 are decay rates (commonly set to 0.9 and 0.999 respectively), and ϵ is a small constant [10].

2.2.4. Normalization

Neural Networks are simply a parametrized function and even though they provide immense insight into various tasks, because of their numeric nature, they are extremely sensitive to how the data is fed. For instance, several works have proved the effectiveness of normalizing the input data before feeding it to the model [21] [35] [10]. In essence, this allows the model to learn the intricate patterns present in the data instead of over focusing on the large absolute differences that can occur.

Normalization is usually done both to the inputs before they are fed to the model, and to the output of each layer in an operation referred to as Batch Normalization. For the inputs, usually standard normalization is used. For Batch Normalization, usually standard normalization followed by trainable re-scale and shift is used.

Given a batch batch normalization proceeds as follows:

Standard Normalization:

$$\begin{aligned} B = \{x_1, x_2, \ldots, x_m\}; \mu_B &= \frac{1}{m}\sum_{i=1}^{m} x_i, \\ \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2, \hat{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2}} \end{aligned} \tag{2.15}$$

Batch Normalization:

1. Apply Standard Normalization to each batch
2. With a trainable γ and β for each batch output x_i the normalized output of the batch normalization layer y_i is calculated in the following way: $y_i = \gamma \hat{x}_i + \beta$

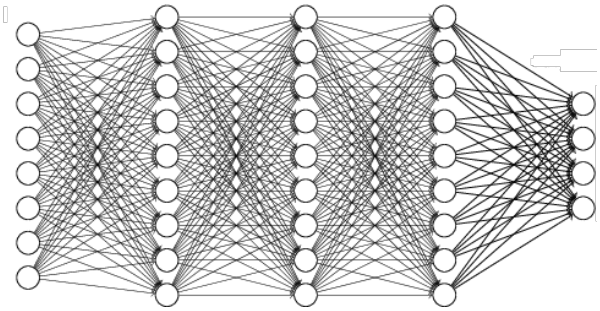


Figure 2.2: General structure of MLP [26]

2.2.5. Multi-layer perceptron

Multilayer Perceptrons (MLPs) refer to neural networks where all neurons in a layer are connected to all neurons in the following layer, as illustrated in Figure 2.7.

We refer to a layer where all inputs are fed to all neurons as a dense layer. A MLP is a model with only dense layers. This comes with advantages - all neurons have access to all information present in the previous layer, all having the capability of extracting precious information from it - and disadvantages - being the most complex layer in terms of number of trainable parameters.

2.2.6. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) address the complexity problem of MLPs by reducing the number of connections from one layer to another. This is done by restricting the connections to a neuron to a subset of spatially close neurons - as illustrated in figure 2.3. This ensures reduced complexity while specializing neurons on localized features [10].

Furthermore, CNNs have increased personalizability. As opposed to MLPs, where the only tunable hyperparameter is the number of neurons, in a CNN, an important aspect is which subsets of neurons are considered at once. This is usually parameterized by kernels, which are a given size, and determine the chunk sizes of the input considered for each neuron of the following layer.

In the case of a one-dimensional (1D) convolutional layer, the operation is applied over an input vector in $\mathbb{R}^n$, where n represents the input length. This layer is typically defined by several key parameters: the number of filters or kernels n_k, the size (or width) of each kernel, and the stride s_k, which determines how many positions the kernel shifts at each step. The convolution operation involves sliding each kernel across the input and computing local dot products, thereby extracting features from different regions of the input, as illustrated by figure 2.3.

The result of this process is a tensor of shape $n_k \times m$, where m is the number of positions each kernel covers, typically approximated as $\frac{n}{s_k}$ under simple assumptions (depends on padding). This output captures n_k different feature maps — typically it is assumed each kernel is extracting different information from the input [10].

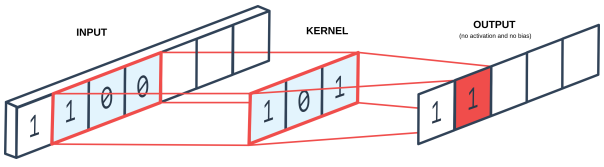


Figure 2.3: Kernel illustration

2.2.7. Language Models

Language Models (LMs) are fundamental tools in natural language processing and one that has been growing in popularity. In essence, they operate on a sequence of words to predict the most likely word to follow. Originally used for tasks such as text generation and translation, language model–inspired frameworks have recently gained prominence in non-language applications, demonstrating

their competence in modeling other sequential structures[23][22][24][29].

At the core of language models is the transformation of discrete inputs into continuous representations through embeddings. Embeddings are functions that map categorical data into dense, real-valued vectors: $\mathcal{E} : \{1, 2, \ldots, C\} \to \mathbb{R}^d$, where d is the embedding dimension. This continuous representation is particularly useful as it enables the model to capture similarities between categories — categories with similar embeddings are treated as similar, while those with embeddings that are close to orthogonal are treated as distinct. Thus, embeddings reveal underlying patterns that might be obscured in a discrete format. This is essential in language generation, as words are discrete objects that have a deep relationship between them which is required to be captured by the model [10].

A language model first converts words (or other discrete elements, such as symbols or categorical units) into unique integer identifiers in a process typically called tokenization. These integer representations are then used as the input to the embedding function $\mathcal{E}$, allowing the model to operate numerically.

Language models often organize tokens into n-grams—contiguous sequences of n tokens extracted from text. Traditional n-gram models estimate the likelihood of a token based on the preceding $n - 1$ tokens.

The ultimate goal of a language model is to estimate the conditional probability

$$P(x_n \mid x_1, x_2, \ldots, x_{n-1}),$$

that is, from a sequence of tokens, to output a probability distribution over the set of tokens that accurately predicts the next token.

2.2.8. Hyperparameter Search

In a model, the weights and biases and other trainable parameters are usually referred to as the model's parameters, while the other variables which regulate the training process or the model's architecture but are not used in the model's computations - like the learning rate and the batch size, are usually refered to as the hyperparameters. Selecting good hyperparameters is key in the learning process of a model. For example, setting a learning rate in a value too high can make the model not diverge due to taking steps that are too big, while setting a learning rate too low can make the model be stuck on a local minimum of the loss function [10].

Hence, it is common in the ML literature, and in particular in the SCA literature, to conduct hyperparameter search to find suitable hyperparameters for a particular model. This consists of selecting a search space to select the hyperparameters from, navigating these with a search algorithm, and using them for a quick training session to evaluate how good they are.

Hyperparameters to be searched for vary, but usually consist of the optimizer, the learning rate, batch size, weight initialization, and also characteristics of the model's architecture.

Common algorithms for conducting hyperparameter search are:

1. **Random Search**
   Random search involves sampling hyperparameter configurations uniformly at random from the search space $\mathcal{H}$, and using it to train until a predetermined number of epochs are completed or a given condition is met. This can be surprisingly efficient when the search space is carefully selected [45]
2. **Grid Search**
   Grid search exhaustively explores $\mathcal{H}$, using each hyperparanmeter set to train until a number of epochs are completed or a given condition is met. Although straightforward and guaranteeing to find the best set, this method becomes unfeasible as the number of hyperparameters increases.
3. **Bayesian Optimization**
   Bayesian optimization (BO) builds a probabilistic surrogate model $P(f \mid h)$ of the objective function $f : \mathcal{H} \to \mathbb{R}$ (e.g., validation loss) and employs an acquisition function $a(h)$ to determine the next hyperparameter configuration $h \in \mathcal{H}$ to evaluate [45].

2.2.9. Knowledge distillation

Knowledge distillation is a process that, as the name suggest, attempts to distill the knowledge acquired by a model into a smaller one, hopefully achieving similar or better results.

Assume there already exists a trained model, which we call teacher network f_T, and it is desired to distill its knowledge into a smaller student network f_S. Denote the logits by $z_T(x)$ and $z_S(x)$ on input x. For temperature $T > 1$, define the softened outputs

$$p_{T,i}(x) = \frac{\exp\big(z_{T,i}(x)/T\big)}{\sum_j \exp\big(z_{T,j}(x)/T\big)}, \quad p_{S,i}(x) = \frac{\exp\big(z_{S,i}(x)/T\big)}{\sum_j \exp\big(z_{S,j}(x)/T\big)} \tag{2.16}$$

$$\mathcal{L} = \alpha\,\mathrm{CE}\big(y,\, p_S(x)\big) + (1-\alpha)\,T^2\,\mathrm{KL}\big(p_T(x)\,\|\,p_S(x)\big) \tag{2.17}$$

where CE is the cross-entropy with true labels y, KL the Kullback–Leibler divergence, $\alpha \in [0,1]$ a balancing hyperparameter, and the T^2 factor corrects for gradient scale differences [14].

The Temperature factor controls the softness of the labels - a value of T close to 0 makes the training data closer to hard labels (distributes the probability heavily towards the most likely category) whilst a Temperature larger than 1 evens the probability distribution, making the labels softer. Both T and α are trainable parameters, and ones for which it is not easy to heuristically select good values.

Empirically, even deep student models can match or exceed their teachers on unseen data, benefiting from the smoother targets and implicit regularization introduced by the softened teacher outputs. Essentially, it relies on the fact that training a model with soft probabilities infused more knowledge into the model (such as that relating to uncertainty) compared to hard classification labels.

2.2.10. Important results in Linear Algebra and Machine Learning

In this section, some important results in Linear Algebra with applications in Machine Learning will be presented. These are results that one way or another were important in some choices made in this work, and that are important to justify some aspects.

Universal Approximation Theorem

Let $K \subset \mathbb{R}^d$ be compact and let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-polynomial, continuous activation function. Then for every continuous $f : K \to \mathbb{R}$ and every $\varepsilon > 0$, there exists an integer m, weights $w_i \in \mathbb{R}^d$, biases $b_i \in \mathbb{R}$ and output coefficients $\alpha_i \in \mathbb{R}$ such that the single–hidden–layer network $N(x) = \sum_{i=1}^{m} \alpha_i\,\sigma(w_i^\top x + b_i)$ satisfies

$$\sup_{x \in K} \big|f(x) - N(x)\big| \;<\; \varepsilon. \tag{2.18}$$

This result, proven in [6], is important as it gives confidence on the methods used - that is - the fact that all functions can be approximated up to an arbitrarily small error with neural networks. Of course, one may need an arbitrarily large number of layers in order to produce an approximation with a small error, but this result is nevertheless important.

Concentration of Inner Products

Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ be two independent random vectors drawn uniformly from the unit sphere $S^{d-1} = \{\mathbf{z} \in \mathbb{R}^d : \|\mathbf{z}\|_2 = 1\}$. Then for any $\epsilon > 0$,

$$\mathbb{P}\left(|\langle \mathbf{x}, \mathbf{y}\rangle| \geq \epsilon\right) \leq 2\exp\left(-\frac{(d-1)\epsilon^2}{2}\right) \tag{2.19}$$

This theorem formalizes the notion that in high-dimensional spaces, random vectors become nearly orthogonal with high probability [44]. This phenomenon is fundamental to understanding the curse of dimensionality, the effectiveness of random projections and dimensionality reduction techniques, and the geometric properties of high-dimensional spaces that underlie many machine learning algorithms.

Eckart–Young–Mirsky Theorem

Let $A \in \mathbb{R}^{m \times n}$ have singular value decomposition

$$A = \sum_{i=1}^{r} \sigma_i \, u_i v_i^\top, \qquad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0.$$

For any $1 \leq k < r$, the matrix

$$A_k \; = \; \sum_{i=1}^{k} \sigma_i \, u_i v_i^\top$$

is a best rank-k approximation to A in both the spectral and Frobenius norms:

$$\min_{\substack{B \in \mathbb{R}^{m \times n} \\ \operatorname{rank}(B) \leq k}} \|A - B\|_2 \; = \; \|A - A_k\|_2 = \sigma_{k+1}, \qquad \min_{\substack{B \in \mathbb{R}^{m \times n} \\ \operatorname{rank}(B) \leq k}} \|A - B\|_F \; = \; \|A - A_k\|_F = \sqrt{\sum_{i=k+1}^{r} \sigma_i^2}. \quad (2.20)$$

This value explains the usefulness of the Singular Value Decomposition, in particular applied to the compact representation of matrices. This can be applied to feature selection for the reduction of the input size of a neural network.

2.3. Symmetric Key Cryptography

A cryptosystem is modeled by three sets:

- $\mathcal{P}$: the *plaintext space*, whose elements are messages m.
- $\mathcal{C}$: the *ciphertext space*, whose elements are ciphertexts c.
- $\mathcal{K}$: the *key space*, whose elements are keys k.

Encryption and decryption are defined as functions

$$E \; : \mathcal{K} \times \mathcal{P} \to \mathcal{C},$$
$$D \; : \mathcal{K} \times \mathcal{C} \to \mathcal{P},$$

such that

$$D\big(k,\, E(k, m)\big) = m \quad \forall\, k \in \mathcal{K},\, m \in \mathcal{P}. \quad (2.21)$$

It is required that inverting $E(k, \cdot)$ to recover m from a given ciphertext c is computationally infeasible without knowledge of the secret key k [42] [15].

Symmetric key cryptography is a class of cryptographic systems in which both the sender and the receiver share the same secret key for encryption and decryption, as opposed to public key, where all agents have a unique key pair - one public (used to cipher messages) and one private (used to decipher messages). The security of symmetric key cryptography relies on the secrecy of the key, and also on an existence of a safe secret sharing algorithm - an algorithm that allows 2 agents to agree on a secret without any eavesdroppers to be able to access it [36].

In symmetric encryption, a plaintext message which is intended to be kept private, is transformed into a ciphertext using the shared key and a defined algorithm. The receiver, who possesses the same key, can reverse the transformation to recover the original message. On the flip side, any eavesdropper that does not possess the secret key cannot recover the plaintext apart from using brute force. Hence, a safe protocol of symmetric key cryptography is a function acting of the space of plaintexts which is easily invertible by anyone who possesses the secret key used, and unfeasible to invert by anyone else. Two factors are, therefore, indispensable: a safe algorithm and big keys.

2.4. AES

The Advanced Encryption Standard (AES) is a symmetric key cipher standardized by the National Institute of Standards and Technology (NIST) in 2001. AES has become the de facto standard for data encryption due to its combination of security and efficiency, being the most widely used.

AES operates on fixed-size blocks of plaintext (128 bits) and supports key sizes of 128, 192, or 256 bits, resulting in 10, 12, or 14 rounds of processing, respectively. Each round consists of a series of operations including substitution, permutation, mixing, and key addition. These operations need to be, on the one hand, deterministic, and on the other hand, strongly key dependent. The design of AES is rooted in algebra over finite fields, offering a strong mathematical foundation that resists linear and differential cryptanalysis. From this point, the AES cipher act described is the one with key size of 256 bits.

AES operates on a 4x4 matrix of bytes called the *state* - initially this is the plaintext organized into a matrix, and gets transformed as it is ciphered. Let $\mathbb{F}_{2^8}$ denote the finite field with 256 elements, typically represented using polynomials with coefficients in $\mathbb{F}_2$ reduced over the irreducible polynomial $m(x) = x^8+x^4+x^3+x+1$. This directly represents one byte and defines the operations to be computed between bytes.

In each of the 14 rounds, the key used is a different one, generated from the initial cipher key using a key schedule, to foster further diffusion.

Each round proceeds as follows:

1. **Key Scheduling** generates new keys for this round
2. **AddRoundKey** - using the bitwise XOR, the round key is combined with the state, creating a new state.
3. **SubBytes**: A highly non-linear function (denominated sbox henceforth) is applied to each. sbox is constructed by taking the multiplicative inverse in $\mathbb{F}_{2^8}$ (0 maps to 0) and then applying an affine transformation over $\mathbb{F}_2$.
4. **ShiftRows** Cyclically shifts the rows of the state matrix to the left by 0, 1, 2, and 3 positions.
5. **MixColumns**: Each column of the state is transformed by multiplying it with a fixed polynomial $c(x) = 03x^3 + 01x^2 + 01x + 02$ modulo $x^4 + 1$ over $\mathbb{F}_{2^8}$. This can be represented as a matrix multiplication:
$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

The final round omits MixColumns, and includes one additional XOR of the state with the last key

Of course, there is no proof that AES cannot be broken faster than by exhaustive key search. In fact, given a plaintext P, a ciphertext C, and a candidate key k, one can verify in time polynomial in the key length whether $\mathrm{AES}_k(P) = C$; therefore, proving that every algorithm requires $\Omega(2^n)$ time to recover an n-bit AES key would yield a worst-case separation of P and NP—i.e. a proof that $\mathrm{P} \neq \mathrm{NP}$, which remains one of the great open problems in theoretical computer science [38] (note, however, that AES has not been proved to be NP hard). Nevertheless, no attack on AES-128, AES-192, or AES-256 is known that improves on brute-force, and its security has been subjected to extensive cryptanalytic study [28].

2.5. Side-Channel Analysis

As mentioned before, the AES, as well as other modern encryption algorithms, is theoretically safe, meaning the best way to retrieve the key when all there is available are plaintexts and ciphertexts, is brute force. However, clever attackers with privileged access to the encryption device may exploit unintended leakages emitted by it. Using this information to attack the device is called a Side-channel attack.

Side-channel attacks come in many shapes and forms, but can roughly be categorized by the nature of the side-channel information it is looking to exploit and the whether or not the attacker has access to a copy of the target device (profiled / non profiled attacks).

As far as the nature of the side-channel information to be exploited is concerned, the attacker can utilize the power consumption [17], the electromagnetic emanations [2], the sound produced [13], the time spent in an operation [18], among others. This work will focus on power consumption and electromagnetic emanation. In these attacks, the leakage comes in the form of an electromagnetic or power signal, which will henceforth be referred to as trace.

Access to a copy of the target device provides the attacker with greater possibilities, as they can study the leakage and construct a strong model before attacking the target. The term non-profiled attack is used to refer to an attack without access to such device, using only the side-channel information obtained from the target device. The main examples of non-profiled attacks are Simple Power Analysis, directly analyzing a single power signal, and Differential Power Analysis, comparing multiple power signals, introduced in 1999 in Paul Kocher's et al seminal paper [17].

On the flip side, profiled attacks refer to the scenario where the attacker has access to an identical device as the one that they intends to attack, therefore being able to construct a framework before attacking. This, therefore, constitutes a more powerful attack. This attack is divided into the profiling phase, where the the clone device is used to build a model that effectively predicts the key based on the signal, and the attacking phase, where this model is employed in analyzing one or several signals from the target device. The attacker is usually assumed to have access to the plaintexts used - operating in a chosen plaintext setting - and to have the goal of retrieving the key. This work focuses on profiling attacks.

For side-channel analysis, a core assumption is that the operations that are being executed when ciphering with a given algorithm in some way correlate with the power signal that is being produced. Notice however that in principle, a certain value will only leak if it is either accessed or changed. Because of this, as the key is not directly accessed in the course of the AES ciphering, it is not expected that the key directly leaks. The common target in side-channel attacks is $sbox(p_i \oplus k_i)$, where p_i and k_i are, respectively, a byte of the plaintext and a byte of the key. With access to $sbox$ (which is public) and to p_i, the attacker can reverse engineer and access k_i. The term label will henceforth be referring to th value $sbox(p_i \oplus k_i)$. In spite of the key having 16 bytes, it is common to focus the attack on one byte of the key - namely the third byte, for reasons that will be explained in the countermeasure section. Hence, the term key refers to the third byte of the key, unless otherwise specified. Several articles state that, even though attacking different bytes vary in difficulty, the same attacks can usually attack all bytes [7] [46]. Hence, recovering the goal of most attacks is to recover a single key byte, as this is a proof of concept that other bytes could probably also be recovered. Furthermore, this leakage produced by an operation can be of various natures. This means that the information about the key that is present on the trace can vary. This brings us to the notion of leakage model.

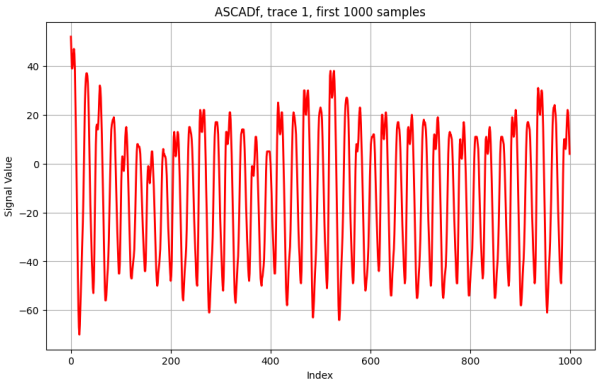


Figure 2.4: First 1000 time steps of the first trace of the ASCADv dataset

2.5.1. Leakage models

In order to structure the attack, the attacker must first select a leakage model, either based on observations or the implementation details of the algorithm. This is what the attacker assumes is being leaked in the signal, and what he will be trying to extract from the analysis of the signal. Commonly used leakage models are the hamming weight model (HW), the hamming distance model (HD) and the identity model (ID). HW models the physical leakage to be correlated with the hamming weight of the target value. HD assumes this correlation to be with the hamming distance between the target value before and after being processed by operations of the cryptographic algorithm. The rationale behind these models is that respectively each 1 bit or bit flip in the leaking operation will correlate with more power usage. ID assumes that the whole target value is leaking in the signal. It is used by obvious reasons - it is the only model that can give hopes of retrieving the key in a single trace.

Hence, in summary, in a profiled attack, the attacker has access to a number N of traces, and trains a model to predict target (Y), as illustrated below:

$$Y_i = f_{leakage}\Big(f_{intermediate\ value}(P_i, K_i)\Big), \quad \forall i \in \{1, \ldots, N\}, \tag{2.22}$$

, where P_i refers to the plaintext, K_i refers to the key, $f_{intermediate}$ refers to the intermediate in AES that is being targeted (usually $sbox(p_3 \oplus k_3)$), and $f_{leakage}$ is the leakage which is the value assumed to be producing in the trace.

2.5.2. Classical Techniques

Power traces and electromagnetic emissions can nowadays be captured with a high frequency, resulting in very large datasets. This gives rise to dimensionality reduction techniques that reduce the number of points and maximize the retained information. These are useful techniques from signal analysis that have been used for a long time in SCA.

Let's assume it is desired to classify the given signal in terms of a certain class $c \in \{c_1, c_2, ..., c_M\}$. This class is usually the leakage value of the key used in producing the signal, that is, either the hamming weight ($M = 9$) or the entire value ($M = 256$). Some points may reveal precious information about this, and other points will contain mostly noise.

Signal to Noise Ratio

Signal to Noise Ratio is a metric computed from a dataset of signals of size N in order to get some information about which points of the signal contains the most information and which points of the signal contains the most noise. This depends on the information we want to extract from the signal. SNR computes a vector of the same size as the original traces, and each value corresponds to the ratio between signal (useful information that discerns in which class a trace belongs) and noise (useless information).

1. Separate the traces into M sets, according to their class;
2. Compute the mean vector of each class $\mu_1, ..., \mu_M$;
3. Compute the global mean vector $\mu = \sum_{i=1}^{M} \mu_i$;
4. Compute variance vector within each class c_i: $\sigma_i^2 = \frac{1}{|c_i|-1} \sum_{t \in c_i} (t - \mu_i)^2$;
5. Compute average variance vector $\sigma^2 = \frac{1}{M} \sum_{i=1}^{256} \sigma_i^2$;
6. SNR=$\frac{\sum_{i=1}^{M} (\mu_i - \mu)^2}{\sigma^2}$.

This way, SNR measures the ratio between 'useful variability' (differences between signals of different classes which we want to differentiate) and 'useless variability' (differences within each class). The points with the highest SNR value corresponds to the points carrying the most information.

Principal Component Analysis:

Principal Component Analysis (PCA) selects the directions of the dataset that maximize the inter-class variance - therefore selecting the principal components of the covariance matrix between each class.

1. Compute the mean vector of each class $\mu_1, ..., \mu_M$;
2. Compute the global mean vector $\mu = \sum_{i=1}^{M} \mu_i$;
3. Compute covariance matrix $S = \frac{1}{M} \sum_{i=1}^{M} (\mu_i - \mu)(\mu_i - \mu)^T$;
4. Compute eigendecomposition of S $S = UDU^T$;
5. Select the largest n eigenvalues from D in absolute value, and corresponding eigenvectors from U, forming U^*;
6. From each trace t, get reduced signal t^* with n points $t^* = U^{*T} s$.

Linear Discriminant Analysis:

Linear Discriminant Analysis (LDA), instead of only maximizing inter-class variance like PCA, maximizes the ratio between inter class variance and intra-class variance, finding V to maximize $\frac{V^T S_b V}{V^T S_w V}$

1. Compute the mean vector of each class $\mu_1, ..., \mu_M$;
2. Compute the global mean vector $\mu = \sum_{i=1}^{M} \mu_i$;
3. $S_b = \sum_{i=1}^{M} N_i (\mu_i - \mu)(\mu_i - \mu)^T$, N_i number of samples of class i;
4. $S_w = \sum_{i=1}^{M} \sum_{j=1}^{N} (s_{i,j} - \mu_i)(s_{i,j} - \mu_i)^T$;
5. Get eigendecomposition of S_b, $S_b = UDU^T \implies S^{\frac{1}{2}} = UD^{\frac{1}{2}}U^T$;
6. Get eigendecomposition of S_w, $S_b^{\frac{1}{2}} S_w^{-1} S_b^{\frac{1}{2}} = VEV^T$;
7. Select the largest n eigenvalues from E, and corresponding eigenvectors from V, forming V^*.
8. From each trace t, get reduced signal t^* with n points $t^* = V^{*T} s$

Recently, with the boom in Machine Learning popularity, many of the classical techniques have been replaced with ML techniques in profiling attacks, as it is the very setting where it is widely used - learning patterns in a training set and gaining the ability to generalize them into new samples. This has shown to produce great advancements in the state of the art. This thesis will mainly focus on the use of Machine Learning techniques. In spite of this, classical techniques still provide precious help in making the job of the ML model easier.

2.5.3. Attack, Evaluation and Metrics

When performing an attack, it is assumed that the attacker has a model, that from an attack trace, produces a probability distribution over the set of possible keys $(p_1, ..., p_{256})$, and one can also compute a guessing vector, that is, an ordering of the keys according to the probability the model gives them $(g_1, ..., g_{256})$, where g_i is the key the model thinks to be the i^{th} most likely to be the correct key. An attacker might have access to $n > 1$ trace samples, so it is paramount to have a way of combining information from different traces. Furthermore, metrics can be separated into 2 classes: those computed by attacking single traces and those resulting from attacking multiple traces and combining the information gathered from them. Addressing the first case, the metrics considered are accuracy - the fraction of traces that have $g_1 = k^*$ and guessing entropy - the average i such that $g_i = k^*$, where in both cases, k^* is the correct key.

One way to evaluate this attack is to compute the accuracy over all attack traces (the fraction of traces in which our model considered the real key as the most likely one). However, classical machine learning metrics like accuracy have shown to be suboptimal when used in SCA scenarios [43]. This is because the attacker does not necessarily need a correct classification of the key in 1 trace, instead, is looking to have a good probability distribution over the set of keys that will allow him to retrieve the keys in the least amount of traces possible. Hence, the need for metrics tailor made for SCA arises.

For metrics using multiple attack traces, independence is usually assumed between key guesses, and the joint probability attributed to key k after n traces is calculated as $P_{k,n} = \frac{1}{Z} \Pi_{i=1}^{n} p_{k,i}$, where $p_{k,i}$

is the probability that the model attributed to key k in trace i and $Z = \sum_{k=1}^{256} \Pi_{i=1}^{n} p_{k,i}$. [31]. After n traces, with the vector $P = (P_{1,n}, ..., P_{256,n})$, one can construct the same guessing vector as before $(G_{1,n}, ..., G_{256,n})$ by sorting the possible keys by the probability given to them. For a successful attack, after considering some number of traces n, one should arrive at $G_{1,n} = k^*$, and this should never change when considering additional traces.

A useful metric used to evaluate a successful attack is N_{tAR} (number of traces until average rank=1):

$$N_{tAR} = \min\Big\{m \in \mathbb{N} \,\Big|\, \forall i \geq m : \forall k \in \{0, 1, \ldots, 256\} \setminus \{k'\},\ \ P_{k',i} > P_{k,i}\Big\}. \tag{2.23}$$

To evaluate a model, several attacks are conducted, with a different set of traces used to attack, resulting in different N_{tAR} values (in case of successful attacks). When referring to a model (and not a single attack), the most widely used metric is N_{tGE}, referring to the number of traces until Guessing Entropy = 1 (maximum N_{tAR} in that set of attacks)

2.5.4. Countermeasures

As the field of SCA developed and new exploits were discovered and developed, also were several countermeasures specifically aimed at SCA. The two main countermeasures that we will consider in this thesis, which are masking and desynchronization.

Masking:

In masking, intermediate values within the cryptographic computation are hidden by random variables. This is done with a a uniformly random mask $m \in \mathbb{F}_{2^8}$, independent of the values $v \in \mathbb{F}_{2^8}$ it is hiding. It is called boolean masking when the used mask is XOR'ed with the sensitive value, and multiplicative masking, when the operation used is field namely multiplication.

This brings us back to the notion that only values handled directly can leak directly in the trace - by not handling sensitive value v, and instead, $v' = v \oplus m$, the values that can leak in the trace are v' and m. Of course, one can still recover v by recovering m and v', but no single point can ever totally leak v. The leakage is said to become second-order, as opposed to first-order.

In the case of AES, masking is usually applied to the output of the SBOX operation using an XOR operation [40]. In particular, in the AES implementation attacked in this thesis, the first two bytes of the key are unmasked and are therefore relatively easy to attack. The third byte is the first masked byte, which is why it is chosen as the target for the attack.

The masking scheme consists of 18 bytes: $(r_1, \ldots, r_{16}, r_{\text{in}}, r_{\text{out}})$, where r_i is specific to each AES round, and r_{in} and r_{out} are used to mask the key being accessed and the SBOX output, respectively. It is important to highlight the operations performed in the first round of AES, as these reveal what is being leaked:

- **Initialization:**

$$state0_i = p_i \oplus r_i, \quad state1_i = r_i \tag{2.24}$$

  leaking the masks

- **Accessing the key:**

$$\begin{aligned} state0_i &= state0_i \oplus (k_i \oplus r_{\text{in}}) \oplus state1_i \\ &= p_i \oplus k_i \oplus r_{\text{in}} \end{aligned} \tag{2.25}$$

  leaking $p_i \oplus k_i \oplus r_{\text{in}}$

- **SBOX table lookup:**

$$state0_i = \text{sbox}[state0_i] = \text{sbox}[p_i \oplus k_i] \oplus r_{\text{out}} \tag{2.26}$$

  leaking $\text{sbox}[p_i \oplus k_i] \oplus r_{\text{out}}$

- **Final step of the round:**

$$\begin{aligned} state0[i] &= state0_i \oplus state1_i \oplus r_{\text{out}} \\ &= \text{sbox}[p_i \oplus k_i] \oplus r_i \end{aligned} \tag{2.27}$$

  leaking $\text{sbox}[p_i \oplus k_i] \oplus r_{\text{i}}$

Each of these steps is leaky and can be exploited by side-channel attacks. The intermediate values typically targeted are $\text{sbox}(p_3 \oplus k_3) \oplus r_3$ and $\text{sbox}(p_3 \oplus k_3) \oplus r_{\text{out}}$.

Before executing the next operation, the output is unmasked. This does not alter the final ciphertext—since the masking will eventually be reversed—but serves only to hide sensitive intermediate values during computation. The masks are generally considered *ephemeral* (used for a single encryption) and *unknown* to the attacker.

Depending on the assumed threat model, it can either be considered that the attacker has access to the random masks during the training phase, or that the masks remain inaccessible even then.

Desynchronization

Desynchronization aims to misalign the temporal structure of side-channel traces. These techniques disrupt the temporal correlation between internal computations and observed leakage by introducing random delays, jitter, or random instruction reordering. In effect, desynchronization transforms the leakage function $L(t)$ into $L(t + \delta)$, where δ is a random variable, making standard correlation-based attacks significantly harder due to misalignment.

In practice, this is done by waiting a random amount of clock cycles where dummy operations are performed, before computing sensitive values, where the maximum waiting time is often fixed and pre-determined.

In this work, when a given dataset is referred to having desynchronization $= d$, it is meant that for every trace, a different $\delta \in \mathbb{Z}, \quad \delta \sim \text{Uniform}(0, d)$ is sampled, and a desynchronized trace replaces the original trace. This desynchronized trace is the same size s as the original one, its first $s - \delta$ values are the same as the values from δ to the end of the original trace, and its last δ are the first δ values of the original trace. Including the final part of the original trace in the beginning of the desynchronized trace mimmicks the dummy operations performed by the device.

2.5.5. Public Datasets

In order to foster easy contribution to the SCA community, and in particular to ML-assisted SCA, several datasets have been made public. These datasets consist of recorded traces of power consumption or electromagnetic emanation, each labeled with the corresponding AES-related information used during the cryptographic operation that generated the signal. This availability enables more systematic and standardized security evaluation. In practical scenarios, such traces could correspond to leakage of cipher operations an attacker might observe if they possess a cloned device identical to the target

The ASCAD database is presented in [2]. The target device is an 8- bit AVR microcontroller (ATmega8515) running a masked AES-128 implementation. Measurements are made using electromagnetic emanation. A boolean masking scheme is used as countermeasure, protecting the sbox computation with a 8-bit random mask. Within the ASCAD database, two distinct datasets exist: ASCADf and ASCADv. They were obtained via the git repository https://github.com/ANSSI-FR/ASCAD .

ASCADf is a dataset with 60000 traces, each with 100000 points. Each trace is labeled with the plaintext, key and masks used in the AES execution that generated it. The key used in all of these executions is the same. A split of 50000/10000 between profiling and attacking traces is suggested.

ASCADv is a dataset with 200000 traces, each with 250000 points. Each trace is also labeled with the plaintext, key and masks used in the AES execution that generated it. A split of 200000/100000 between profiling and attacking traces is suggested. The key used is random in the profiling traces, and fixed for the attacking traces (to allow for an attack using multiple traces), therefore, this is a harder dataset to attack.

On top of this, the authors have studied the SNR values of several masked and unmasked values among the trace. The SNR of unmasked values is very close to 0, which means that the masking is successful and these values are not directly leaking. A time interval where SNR is highest is indicated, as a suggestion of dataset to use. This interval has 700 and 1400 points, for, respectively, the ASCADf and the ASCADv datasets.

As suggested in [30], I will refer to this smaller dataset of 700 or 1400 points as OPOI (optimized points

of interest). Notice that using OPOI indirectly assumes the attacker has access to the masks in the training phase, in order to compute SNR of the leaky values. I will refer as NOPOI (non-optimized points of interest) to the dataset of the raw traces when subsampled with average pooling with a kernel of size 40 and 50% overlap, as suggested in [30].

Also utilized in the literature are techniques of LDA analysis to extract more informative features. This approach, referred to as Refined Points of Interest (RPOI) in [30], assumes the attacker has access to both secret shares - namely, the intermediate value $sbox(p_i \oplus k_i) \oplus r_i$ and the mask r_i - during the profiling phase. RPOI are selected from the highest SNR peaks of these two signals (similarly to the OPOI case, but not necessarily consecutive points). LDA is applied for dimensionality reduction, producing a total of ten features (this number can vary, but ten is suggested in [30]). Because of the strong assumptions and small resulting size, this feature selection method is more useful in classical SCA and not in deep learning, and is not considered in this work.

3

Related Work

As security grows in importance, naturally, side-channel analysis is a research field which has a good amount of research devoted to it. It is therefore paramount to investigate the work that has already been developed and the gaps that can be filled in the existing literature. As side-channel analysis is a broad field of research with many different exploits used to attack devices, the focus for this work will be on Power Attacks and Electro Magnetic attacks. Hence, let the term side-channel attack/side-channel analysis be referring to these exploits from here on.

This chapter will provide an overview of the existing literature on side-channel analysis, both giving a historical framing by mentioning the most important papers throughout the years, but also providing the most important state of the art results. It is important to mention that this literature review is not meant to be a complete review of the research done in SCA - it will be mostly focused on applications to the ASCAD dataset and on deep learning methods, as these are the results most easily comparable to the ones achieved in this thesis. Nevertheless, some results obtained by traditional side-channel analysis will be presented to highlight the performance of deep learning in this scenario. Furthermore, as this thesis features an application of a LLM-like model to a non-language related setting, also some motivating results will be presented that showcase the effectiveness of these frameworks to settings other than language processing. Finally, this thesis will be framed among the existing literature, by identifying the research gap which it intends to fill. The main search engine used to find literature was google scholar, and the search conducted started from papers recommended by my daily supervisor and expanded by navigating the papers they referenced, investigating the different approaches being taken and searching for the state of the art.

3.1. Historical Framing and Classical Side-Channel Analysis

The inception of Side-channel analysis as a research field can be attributed in large part to Paul Kocher et al's seminal paper [17], where Differential Power Attacks were first introduced. In this work, the power consumption traces recorded during encryption operations are divided into two groups based on the value of a specific bit computed by the device in an intermediate AES operation. By averaging the traces in each group and computing their difference, the differences in the leakages happening in each group emerge, which would be compared to the leakage happening when attacking. This breakthrough demonstrated that even small, data-dependent fluctuations in power usage—previously dismissed as mere noise—could be exploited to reveal secret cryptographic keys.

Template attacks [41] model the trace signals as a multivariate Gaussian distribution and infer the most likely average signal and covariance matrix for each trace class (either the *label* or its hamming weight, depending on the leakage model chosen), based on the available dataset. During the attack phase, for a given trace, the class that attributes the highest probability of obtaining the trace is selected.

Later, Correlation Power Analysis was conceived as an improvement over DPA. Here, a predicted leakage vector is calculated and the correlation between the observed leakage and the predicted leakage is computed for each class. The class with the highest correlation is selected as the most likely one.

Finally, another common attack using classical statistics is the Stochastic Attack, where the power consumption is modeled as a stochastic process. The observed trace is considered as a random variable affected by both the leakage and noise. By maximizing the likelihood or, in a Bayesian framework, by computing the posterior probability $P(k|x_1,,x_N)$ with appropriate priors, the attacker aggregates evidence across many traces to recover the key. This appears to be a superior attack to the other ones mentioned in this section when the number of samples available for profiling is low [9].

Template attacks combined with Linear Discriminant Analysis have shown to be capable of attacking the ASCADv dataset with less than 20 traces [5].

3.2. Deep Learning in Side-Channel Analysis and State of the Art

Similarly to most Data Analysis problems, specific techniques are being surpassed by Deep Learning, and currently, the best results that are obtained in public dataset utilize DL [30].

With the publication of the ASCAD database, in the paper that introduces the dataset [2], the authors offer other contributions. Firstly, they analyze the leakage of different interesting values that is found in the traces. Specifically, they compute the snr values for $sbox(p_3 \oplus k_3)$, $sbox(p_3 \oplus k_3) \oplus r_3$, $sbox(p_3 \oplus k_3) \oplus r_{out}$, r_3 and r_{out}. As expected, $sbox(p_3 \oplus k_3)$ has SNR constantly close to 0, while the other values leak. Furthermore, they present a comparison between a template attack, a CNN and a MLP in the scenario of attacking the ASCAD fixed key dataset with and without desynchronization. The MLP resulted from a rough hyperparameter search and is composed of 6 dense layers with 200 neurons each using the ReLu activation function. A similar search was conducted for the CNN, resulting in a model with 5 convolutional layers followed by big dense layers with 4096 neurons. The template attack was executed on 10 or 50 features, depending on the level of desynchronziation, which were extracted by PCA. In the case where desynchronization is set to 0, the Template Attack was the most successful, managing to achieve guessing entropy 0 in 450 traces, compared to the near 1000 needed by the CNN and the MLP. However, as desynchronization was raised to 50 and 100, the CNN model proved to outperform the other 2, followed by the MLP and the Template attack. In spite of this, none of the models managed to achieved guessing entropy 0 in the 5000 traces used for attack. The CNN used over 60 million parameters.

These results showcased that Deep Learning models seem to be much more resilient to desynchronization countermeasures than the Template Attack. However, these results are still clearly unconvincing.

Whilst CNNs are regarded as the most resilient method to desynchronization, in [19], the resilience to this countermeasure is said to be more dependent on dataset augmentation - generating various desynchronized samples with different levels of desynchronization from the same original sample- than on the architecture itself.

Better results with simpler models are achieved in [47], which interestingly searches for good architectures and hyperparameters not through a search algorithm, but through employing visualization tools and heuristically selecting the optimal architecture from the information obtained from them. The role of hyperparameter is evaluated using Weight Visualization, Gradient Visualization and Heatmaps.

In this paper, the network is tuned from dataset to dataset, allowing for a network tailor-made for the task's needs, which allows for smaller networks. With no desynchronization, it achieves guessing entropy 0 after 191 traces, and needs 270 traces with desynchronization of 100. The trainable parameters are significantly reduced compared to the previous work, to 16960 and 142044 with 0 and 100 desynchronization, respectively.

This paper is quite relevant to this work especially because the code was made fully public and because the models were extremely simple, making them a great start point.

For synchronized traces, they recommend using only one convolutional block and minimizing the length of the filter, and for desynchronized traces, an architecture that enables the network to focus its feature selection on the PoI's.

Other works employ a more systematic approach to hyperparameter and architecture search, switching from visualization techniques to a search algorithm.

For example, [32] attempts to find a good model, composed of a CNN followed by dense layers, in a different way. It performs search using a Markov Decision Process where the Markov states are CNN characteristics, and searches for the best hyperparameters with Q-learning. All aspects of the architecture were searched (number of layers, number of dense layers, size of layers, padding, layer depth, kernel size, stride, pooling activations). It does this guided by 2 different reward functions: one simply attempting to reach the best performance and another which also penalized the size of the network, leading to smaller models. With no desynchronization, the networks have, respectively , 79439 and 1282 trainable parameters, and achieve rank 0 in 202 and 242 attack traces, with similar results with 100 desynchronization (slightly worse performance on both cases). On the ASCAD variable dataset, the amount of parameters used was similar, and the number of attack traces needed was 490 and 1018 respectively. This showcases that small networks are extremely competent, and that an improvement of results in the literature is usually accompanied by a decrease in network complexity. This is especially true for the ASCAD fixed dataset, as the performance on the ASCAD variable dataset was significantly worse for the small network. When considering the Hamming Weight leakage model in the fixed dataset, the performance of the small network ended up being better. They also do a comparison with random search achieving practically equal outcomes, in spite of highlighting that random search is highly dependent on the pre-defined search space.

Both Bayesian optimization and random search are employed in [45] with the same search space. The results suggest that Bayesian Optimization works well, but random search can also find excellent profiling models, actually surpassing the performance of Bayesian Optimization, unless the search space is too large, in which case random search struggles to find decent hyperparameters. This entails that in order to use random search, a pre-selection of reasonable ranges needs to happen. If this pre-selection is successful, random search outperforms Bayesian Optimization. On the fixed key dataset, also using OPOI, the random search found a MLP that managed to achieve guessing entropy 0 after 80 attack traces. When searching the architecture for a CNN, the search space considered was too large and it did not find a good model (in a CNN scenario there are many more tunable aspects of the architecture compared to an MLP). BO found a CNN that achieved guessing entropy 0 after 158 traces. The paper also does a comparison in which metric to use for searching the model. Results are mixed, and the only surprise is that accuracy, which is not considered to be a good measure for SCA scenarios, especially when in presence of a class imbalance in the HW leakage model sceneario, performed just as good as the other measures.

In the paper [30], an extremely broad analysis is done over 4 datasets including ASCAD fixed, ASCAD variable. Also comparisons between using OPOI, NOPOI and RPOI are done, using both HW and ID leakage models. Apart from this, the authors compare Grid Search and Random Search for an MLP and a CNN followed by dense layers. The considered search space was considerably smaller for grid search compared to random search. A different search was conducted for each dataset. Each search went through 500 different models, training each for 100 epochs and evaluating afterwards. As these results are especially good and complete, I present them here in full. These constitute state of the art in many of the datasets, and this work is one of the main references both for my work and also for the SCA field:

Hyperparameter	**Random Search**	**Grid Search**
Optimizer	`Adam`, `RMSprop`	`Adam`, `RMSprop`
Dense Layers	1, 2, 3, 4, 5, 6, 7, 8	1, 2, 3
Neurons	10, 20, 50, 100, 200, 300, 400, 500	20, 50, 100, 200
Activation Function	`SeLU`, `ReLU`	`SeLU`, `ReLU`
Learning Rate	1e-3, 5e-3, 1e-4, 5e-4	1e-3, 1e-4
Batch Size	100 to 1 000 (step: 100)	200, 400
Weight Initialization	`random`, `glorot` or `he uniform`	`glorot`
Regularization	`None`, `Dropout`, `l1` or `l2`	`None`
Total Search Space	**5 971 968**	**192**

Table 3.1: Search space for MLP in [30]

Hyperparameter	Random Search	Grid Search
Optimizer	`Adam`, `RMSprop`	`Adam`, `RMSprop`
Convolution Layers	1, 2, 3, 4	1
Convolution Filters	$4 * 2^{i-1}$, $8 * 2^{i-1}$, $12 * 2^{i-1}$, $16 * 2^{i-1}$ (i = conv. layer index)	5, 10
Convolution Kernel	26 to 52 with a step of 2	2, 4
Convolution Stride	Convolution Kernel / 2	1
Pooling Type	`maxpooling`, `avgpooling`	`avgpooling`
Pooling Size	2, 4, 6, 8, 10	2
Pooling Stride	Pooling Size	2
Dense Layers	1, 2, 3, 4	1, 2
Neurons	10, 20, 50, 100, 200, 300, 400, 500	50, 100
Activation Function	`SeLU`, `ReLU`	`SeLU`, `ReLU`
Learning Rate	1e-3, 5e-3, 1e-4, 5e-4	1e-3, 1e-4
Batch-Size	100 to 1 000 (step: 100)	200, 400
Weight Initialization	`random`, `glorot` or `he uniform`	`glorot`
Regularization	`None`, `Dropout`, `l1` or `l2`	`None`
Total Search Space	**>1B**	**128**

Table 3.2: Search space for CNN in [30]

Dataset	Neural Network Model	Feature Selection Scenario	Amount of POIs (HW/ID)	Attack Traces (HW/ID)	Search Success (%) (HW/ID)	Trainable Parameters (HW/ID)
`ASCADf`	MLP	RPOI	200/100	5/**1**	99.22%/96.86%	82 209/429 256
`ASCADf`	CNN	RPOI	400/200	5/**1**	99.23%/99.08%	499 533/158 108
`ASCADf`	MLP	OPOI	700/700	480/104	82.80%/68.80%	16 309/10 266
`ASCADf`	CNN	OPOI	700/700	744/87	55.53%/35.33%	594 305/62 396
`ASCADf`	MLP	NOPOI	2 500/2 500	7/**1**	74.50%/39.00%	2 203 009/5 379 256
`ASCADf`	CNN	NOPOI	10 000/10 000	7/ **1**	15.40%/2.45%	545 693/439 348
`ASCADf`	CNN	NOPOI desync	10 000/10 000	532/36	2.44%/2.64%	268 433/64 002
`ASCADr`	MLP	RPOI	200/20	3/**1**	99.23%/100%	565 209/639 756
`ASCADr`	CNN	RPOI	400/30	5/**1**	100%/100%	575 369/636 224
`ASCADr`	MLP	OPOI	1 400/1 400	328/129	71.40%/37.25%	31 149/34 236
`ASCADr`	CNN	OPOI	1 400/1 400	538/78	47.92%/23.95%	270 953/87 632
`ASCADr`	MLP	NOPOI	25 000/25 000	6/ **1**	44.39%/7.02%	5 243 209/12 628 756
`ASCADr`	CNN	NOPOI	25 000/25 000	7/ **1**	19.17%/4.35%	369 109/721 012
`ASCADr`	CNN	NOPOI desync	25 000/25 000	305/73	0.71%/1.04%	22 889/90 368
`CHES_CTF`	MLP	OPOI	4 000/4 000	27/1 905	54.24%/0.09%	1 383 609/213 106
`CHES_CTF`	CNN	OPOI	4 000/4 000	462/>3 000	2.99%/0.00%	666 429/593 780
`CHES_CTF`	MLP	NOPOI	3 750/30 000	8/13	69.75%/11.11%	198 209/6 091 856
`CHES_CTF`	CNN	NOPOI	7 500/7 500	238/>3 000	7.89%/0.00%	216 673/1 319 552
`CHES_CTF`	CNN	NOPOI desync	7 500/7 500	248/906	12.22%/3.05%	374 233/564 436
`DPAV4.2`	MLP	RPOI	900/100	3/**1**	100%/95.25%	956 009/287 956
`DPAV4.2`	CNN	RPOI	700/200	8/**1**	93.05%/97.54%	34 709/697 112
`DPAV4.2`	MLP	OPOI	800/800	11/**1**	93.33%/71.62%	48 159/251 856
`DPAV4.2`	CNN	OPOI	800/800	31/170	33.33%/2.63%	1 158 857/424 956
`DPAV4.2`	MLP	NOPOI	15 000/3 750	400/2 577	5.49%/0.14%	1 501 009/1 603 056
`DPAV4.2`	CNN	NOPOI	3 750/15 000	2 767/>3 000	0.16%/0.00%	578 033/89 384
`DPAV4.2`	CNN	NOPOI desync	3 750/3 750	>3 000	0.00%/0.00%	34 836/72 677

Table 3.3: Results from [30]

In all of the mentioned works, the target for the deep learning models is the usual $sbox(p_3 \oplus k_3)$. However, this value does not leak - instead, all leaky values are masked, and selecting those as target would imply that the attacker knows these random nonces used. To address this, [25] builds an architecture that guesses the mask and the masked sbox value separately, and these are trained either explicitly or implicitly. In essence, supposing the attacker does not have access to the masks, it still can infuse the knowledge of the existence of masking into the model, achieving better results.

The fact that there are several leaky intermediate variables that can retrieve the key is leveraged in [4] a model that utilizes all of this information was built, combining it by what is called belief propagation - essentially multiplying the probability distributions obtained by exploiting each leaky variable (product of experts).

Finally, [46] presents a novel architecture that instead of outputting a probability distribution over the

256 possible values, it outputs a probability distribution over 2 values for each of the 16 bits of the attack byte. Hence, it is essentially an architecture that is resilient to all bit-wise leakage models. It managed to achieve guessing entropy 0 after 19 and 898 traces, in the ASCAD fixed and variable, respectively.

3.3. Large Language Models in Non-Language Settings

Language models' primary application is to interpret and/or generate text. Its key feature are word embeddings, which allow words to be put into context by learning a translation to a high dimensional vector space. Although mainly used for language related tasks, language models have found applications outside the scope of language generation/interpretation. The key for this is to treat certain information as one would words in a classical language model application.

As a motivation and proof of concept,here some works are presented that did exactly this. These are merely examples and not in any way an extensive list.

Protein engineering is posed in [24] as an unsupervised sequence generation problem, where tokens can either be an amino acid or a conditioning tag, and uses a Language Model to predict and generate the protein's sequence.

Time series have also been subject to LLM's in [29]. It segments into patches as input tokens to Transformer, using these tokens to predict the future behavior of the time series.

A pre-trained language model has also been applied to an interactive decision-making task, by tokening all the necessary inputs - the goals, the history, and the observations, and attempting to predict the optimal actions to take when faced with this situation [23].

Finally, [22] presents a model which learns from a vocabulary of audio tokens to generate music.

All of these achieve good results, sometimes even state of the art, in their field, hence proving that Language models are not restricted to tasks directly linked to language.

3.4. Large Language Models in Side-Channel Analysis

Language models have been widely used in Acoustic Attacks - these are attacks that exploit, for example, the noise made from a keyboard to reconstruct the text that was typed [13]. This task is directly related to language, so despite interesting, it is not directly related to this thesis.

To the best of my knowledge, only 2 works have employed language models to side-channel attacks.

Firstly, [8] aimed to replace Points of Interest selection and dimensionality reduction by utilizing word embedding, which converts power traces into sensitive vectors. Hence, the application of the language model was directed to the analysis of the trace itself, namely in the preparation of this data. Here the power value corresponding to each sampling point in a fixed clock is vectorized, and the word vector is used to replace the power. Following this is a Long Short term memory, and a usual CNN or MLP network is applied to the output.

The other one is [20]. It presents Order vs Chaos, a language-model approach for side-channel attacks combining the strengths of multitask learning (via the use of a language model), multimodal learning, and deep metric learning. Instead of using embeddings to represent the trace, it uses the information about the AES execution as tokens, namely, the plaintext, the key, and the label. It is important to note that there are no attention mechanisms in play, and that the goal of the network is not to predict the "most likely next word" - the main purpose of the embeddings is to infuse domain knowledge into the neural network. Hence, while it is not a real language model, because of the resembling architecture, I refer to it as a Language Model-inspired framework. It managed to achieve a guessing entropy of zero with 1 trace on ASCADf, both with OPOI and NOPOI, and on ASCADv it achieved a guessing entropy of zero with 1 and 61 traces, for the NOPOI and OPOI cases, respectively.

3.5. Research Gap and Thesis Proposal

Here are some of the research questions that the literature has not yet approached in depth and that I plan to study in this thesis:

- What performance can a language model-inspired framework reach in the ASCAD dataset?
- What architecture works best for this task?
- What assumptions does it need to be successful?
- How resilient to desynchronization countermeasures can a language model-inspired framework be?
- How do different feature selection methods impact the performance of a language model-inspired framework?
- What is the most effective way to conduct the training process of a language model-inspired framework?

[20] will be the baseline for this work. Although its architecture is innovative and its results are extremely good, there are several aspects which make it an incomplete work. The main one is that the code is not public and the description of the implementation is not concrete, making it unreproducible and most choices are justified with ”heuristics”. Furthermore, the work assumes knowledge of the masks during the training process and is not faced with desynchronization. These two aspects are important to understand the applicability of LM’s in SCA and that I will be addressing.

In this thesis I present a model with the same skeleton as the one proposed in [20]. The specific deep learning architecture used in each model block will go through hyperparameter search independently of the results in [20]. So, even though the model block names are kept the same, their architectures are different.

Furthermore, once having built this model, there are several experiments which I plan to do to expand the result space and deepen the understanding of how useful can language models be in side-channel analysis, therefore investigating the answers to the research questions. Here is a rough roadmap of the work to be developed:

- Developing a general architecture for the model, called Sane or Silly (SoS), that may or may not have access to mask knowledge during the profiling phase;
- Conducting search in the architecture of each model block;
- Conducting search in the hyperparameters used in training;
- Experimenting with feature selection;
- Experiment with desynchronization;
- Experiment with different silliness generation methods;
- Experiment with knowledge distillation.

4

Methodology

4.1. Sane or Silly

In a typical standard classifier, the input consists of measurements (X), and the output corresponds to a label (Y), as detailed in section 2.5.1:

$$X_i,\ Y_i, \quad \forall i \in \{1, \ldots, N\},$$

with:

N: The number of traces used for profiling.

X: A set of N traces, each corresponding to a signal of a certain physical leakage during a AES execution

Y: A target vector of length N with information about each trace that is to be attacked - - usually the value $sbox(p_3 \oplus k_3)$.

The goal is to create a model that approximates as closely as possible a function $f : X \to Y$ such that, for the given dataset, $f(X_i) = Y_i$.

In this work, a different approach is taken. The input consists of measurements (X), and metadata about the operation taking place (M). This metadata consists of the plaintext being ciphered, the intermediate value label and the key used. In the case of the dataset ASCADf, information about the key is not included, as the key is fixed, hence providing this information would make this task trivial. Therefore, in ASCADf, the metadata consists only of plaintext and label, whereas in ASCADv, information about the key is added. This metadata can either be Sane - corresponding to values present in the dataset, or Silly - corresponding to values that do not correspond to the real ones. The goal of Sane or Silly is to judge, from a trace and its corresponding metadata, whether this metadata is Sane (the real values) or Silly (not the real values). It is important to note that there are several different ways to create these Silly samples. Whilst different ones were experimented with, the plaintext was always left unchanged, and the label value was always generated according to the plaintext and the key (either sane or silly). Hence, both for sane and silly samples, the combination of key, label and plaintext was possible - but in sane samples, contrarily to silly ones, it corresponded to the actual values used in the AES execution that generated the leakage.

For all $i \in \{1, \ldots, N\}$, the input to the model is the 3-gram together with the trace:

$$(P_i, K_i, sbox(P_i \oplus K_i); T_i)\,, \tag{4.1}$$

and its silly counterpart:

$$\left(P_i, \ddot{K}_i, sbox(P_i \oplus \ddot{K}_i), T_i\right), \tag{4.2}$$

where:

- P is the third byte of the plaintext that was ciphered in the AES execution that generated the leakage;
- K is the third byte of the key that was used in the AES execution that generated the leakage;
- $\ddot{K}$ is a byte value that does not correspond to the real one;
- T represents the side-channel measurement.

The output corresponds to a label $L \in [0, 1]$ - corresponding to the probability that the model attributes to the event of the metadata being Sane. The dataset no longer has N samples - it has N Sane samples and a tunable amount of Silly samples (for training, a 1:1 ratio of Sane to Silly samples was maintained, but this ratio can be tuned as it is definitely an important aspect of training).

Sane or Silly is made up of 5 blocks, as shown in Figure 4.1.. These are:

- the pre-trained mask predictor (PTMP);
- the metadata embedding (ME);
- the language model (LM);
- the preprocessor (PP);
- the deep metric learner (DML).

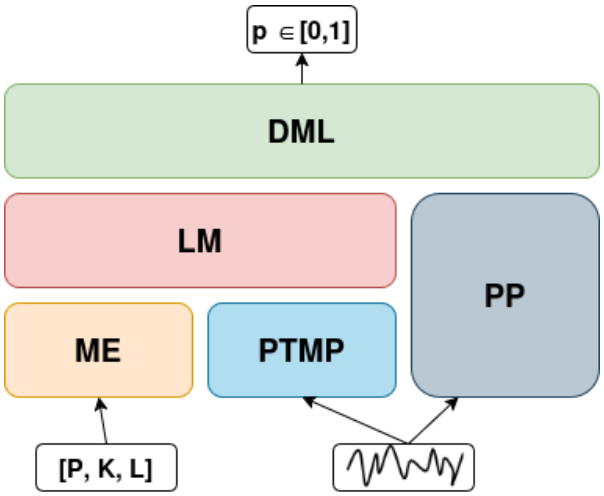


Figure 4.1: SoS illustration

The pre-trained mask predictor (PTMP) is a pre-trained neural network with the goal of predicting the masks. It takes the trace as input and outputs a probability distribution over the possibilities for the used mask (256). This design choice is based on the fact that knowledge of the mask is essential to retrieve key-related information (as all leakages are masked), however, it is not known when attacking. Therefore, this information is as important as the information present in the metadata, and essentially could be part of the metadata. This, however, would translate into a significant increase in complexity for a gain that would not be significant, as the goal is not to predict the masks. Experiments were run with this block predicting the mask r_3, the mask r_{out}, and both, as knowledge of one of these suffices to retrieve the key, but knowledge of both can streamline the learning process. The results using both masks in PTMP will be compared to the results of using each mask separately, and combining the results afterwards similarly to what is done in [4]. It is important to note that using a block that is pre-trained on predicting the masks is not only inconvenient (adding extra steps to the training process), but also it assumes mask knowledge during the profiling phase, placing the attacked in a white-box threat model. To address these issues, experiments were conducted where this block was not trained explicitly, but instead, as part of the entire network, whilst keeping its shape, as done in [25]. Therefore, instead of a PTMP, in these cases, a NTMP was employed.

The metadata embedding, as suggested by the name, applied embeddings to the grams originated by the metadata. These grams are generated by taking the 3 input values, (plaintext, label, key), which are all integers in $\{0, 1, \ldots, 255\}$ and turning them into a sequence of unique words (plaintext, label + 256, key + 512). Hence, this is analogous to a language model, where, for example, the gram 611 corresponds to a word with meaning "key = 99".

The language model takes as inputs the embedded metadata and the mask knowledge (probability distribution over the masks). Hence, with a good mask predictor, the language model has full knowledge about the AES operation, and simply combines it and processes it representing it in the optimal way for the model, outputting a vector with variable shape.

The preprocessor takes as input the trace and analyzes it, outputting a vector of the same size as the output of the language model. Its purpose is similar to the language model, to process the trace signal in order for it to be optimally represented before it gets get to the deep metric learner.

Finally, the deep metric learner receives as input 2 vectors of the same size, one from the language model, containing knowledge about the metadata provided which should be optimally represented, and the preprocessor, containing knowledge about the metadata leaking in the trace which should also be optimally represented. In this block, these 2 vectors are compared, with the goal of confirming whether or not the information given by the language model corresponds to the information extracted from the trace by the preprocessor. These 2 vectors are processed in parallel, with shared weights, and the output is again 2 vectors of equal size, to be compared with cosine similarity. A cosine similarity close to 1 means that the outputted vectors have similar directions, and in this model, corresponds to matching information from the language model and the preprocessor, and therefore the values in the grams being correct. A cosine similarity close to 0 means opposite directions, and therefore not matching. A plausible observation here would be that, whilst expecting matching information to result in similar directions in their representation seems reasonable, expecting not matching information to result in opposite directions seems too much, considering there are many possible wrong values. Because of this, it is important to work in relatively high dimensions, to have many near-orthogonal directions (as highlighted by the Concentration of Inner Products). This block is composed of parallel dense layers with shared weights.

The general idea and purpose of this design is that in essence, we infuse the model with all the information about the operation taking place, therefore making it more informed. With this, he judges whether or not the trace matches the information provided. Infusing the model with the metadata should make it more able to analyze the leakage taking place, potentially resulting in better analysis of the trace compared to a standard classifier.

There are 4 different baseline versions of SoS - corresponding to the differences in PTMP: these are using r_3, using r_{out}, using both, using NTMP.

In spite of referring to the model as a language model, one important part of language models, the attention mechanism, is not present. This is because in this scenario, context is not particularly important - the value of a plaintext does not depend of the value of the key - and the relations between these and label will be learned by the rest of the model.

Details about the specifics of the blocks, as well as the searchable parameters are provided in section 5.3., and the concrete description of the networks used in each scenario is provided in the appendix.

4.1.1. Silliness Generation

Because SoS's task is simply discerning Sane samples from Silly samples, one aspect about the training process that can be experimented with is the Silly dataset. There are different aspects about it that could make a difference in the effectiveness and speed of SoS's training:

- When Silly samples are generated;
- How Silly samples are generated;
- What ratio of Sane and Silly samples is employed.

Concretely, one can decide to generate the Silly samples upfront and use the same dataset throughout the entire training process or change it every n epochs, ensuring diversity in the data that the model

sees.

If the Silly dataset is generated upfront, the only reasonable way to generate it is randomly. However, if data is generated not upfront but throughout the training process, one can generate it by investigating what values cause the model to make the biggest errors and generate a new dataset using those. This is a technique called hard negative mining, and has shown good results in several machine learning applications [37].

Lastly, whilst the real ratio between Sane and Silly samples are 1 Sane for 255 Silly samples, because when attacking, the probabilities are normalized, there is no strict need to employ that same ratio when training.

In this work, a ratio of 1:1 between Sane and Silly samples was always used this is justified by simplicity and memory constraints. Experiments were conducted with 3 different Silliness generation scenarios: generating the data upfront, generating random Silliness every epoch, and generating Silliness with hard negative mining every n epochs. n is definitely a parameters that strongly impacts the training process. $n = 1$ ensures maximum diversity, but seriously slows down the training session, as for a big dataset like ASCADv, generating entirely new 200000 samples is time consuming. Different values for n were experimented with, and $n = 3$ ended up being chosen, but it is important to note that there could be more optimal choices, and that this choice clearly depends of the amount of computational resources available.

4.1.2. Bias Removal

Framing a multi-class classification problem into a binary one can cause some problems. Namely, it is common that biases arise in the models towards some keys [12]. This is because in a multiclass classification problem, as long as the training data is balanced, the model recieves somewhat directly the information that all classes are equally likely (all key bytes, in this case, are assumed to be equally likely). However, when the input is given as a trace paired with a key possibility, not only does this information become more convoluted, but also there could be an imbalance in the amount of times the keys are used in the dataset as Silly samples (this was not kept uniform). Furthermore, the distribution is wildly different in the profiling and in the attack phases - while in the profiling phases there is a 1:1 ratio between Sane and Silly Samples, in the attacking dataset the model is trying to pick the one Sane sample in a set of 256 samples.

After running some trained models, particularly on the ASCADv dataset, this problem was observed. As a result, the models often became strongly biased toward predicting certain keys - different from model to model.

It is important to note that this is not expected behavior - because the plaintexts and keys change with every sample, the sbox function is highly non-linear, and the leaking value is $sbox(p_3 \oplus k_3) \oplus r_3$. When the key is fixed (as in the attacking dataset), there should be no key (apart from the correct one) that is consistently preferred by the model. This kind of bias would never happen in a standard classifier; it arises due to the model interpreting the metadata fed to it.

This issue was frequently observed. After examining the model behaviors, it was found that all models exhibited some degree of bias toward certain keys - which varied from model to model. To address this, two ”debiasing” methods were experimented with. Whilst several other more complex ones could be used, the simpler methods showed superior performance in preliminary testing, so only these two were considered:

Prior-Based Debiasing

Inspired by prior correction methods in [27], the model was evaluated on a balanced large (40k samples) training subset with an equal number of samples per key. For each key i, we computed the total score:

$$P_i = \sum_{j=1}^{N} p_{i,j},$$

where $p_{i,j}$ is the probability the model assigned to key i on sample j, and N is the number of samples

in this balanced, large subset. The outputs were debiased by scaling inversely to the key priors:

$$\tilde{p}_{i,j} = \frac{p_{i,j}}{P_i},$$

followed by normalization across all keys for each sample. This ensures the model treats all keys as equally likely.

Model Combination

Combining the outputs of two models can also reduce bias and improve generalization, as two models usually are observed to have different biases. Consider $p_i^{(1)}$ the probability model 1 assigns to key i and $p_i^{(2)}$ the probability the model 2 assigns to key i. $\hat{p}_i$, the joint probability of key i can be computed in several ways:

- **Product Rule (Independent Models):** Assuming independence, combine probabilities by multiplication:

$$\hat{p}_k = \frac{p_k^{(1)} \cdot p_k^{(2)}}{\sum_j p_j^{(1)} \cdot p_j^{(2)}}. \tag{4.3}$$

- **Loss-weighted Averaging:** Given validation losses $\mathcal{L}_1$ and $\mathcal{L}_2$, combine using:

$$\hat{p}_k = \frac{\frac{1}{\mathcal{L}_1} p_k^{(1)} + \frac{1}{\mathcal{L}_2} p_k^{(2)}}{\frac{1}{\mathcal{L}_1} + \frac{1}{\mathcal{L}_2}}. \tag{4.4}$$

This weights models by reliability, favoring lower-error predictions [27], and no longer relies on the independence assumption.

4.2. Training Strategy and Optimization

The pipeline followed in training this model was

- Hyperparameter search for the PTMP (for r_3 and r_{out}), searching over training hyperparameters and some aspects of the model's structure;
- Training and evaluation of the PTMP with the found hyperparameters;
- Knowledge distillation with the best PTMP as the teacher model;
- Hyperparameter search for the 4 versions of SoS, using the best found PTMP, searching over training hyperparameters and some aspects of the model's structure.
- Training and evaluation of the 4 versions of SoS with the found hyperparameters, with appropriate debias methods;
- Hyperparameter search, training and evaluation for the 4 versions of SoS with different Silliness generation methods;
- Hyperparameter search, training and evaluation for the best SoS version(s) against different levels of desynchronization, compared with fine tuning of the previously found models for this task.

This procedure will be conducted for every dataset considered - each different setting uses different hyperparameters and is trained separately. With this, it is relevant to highlight that it is not intended to search for a general framework, only to evaluate the capabilities of this network if tuned for each setting.

The different settings considered include different feature selection methods (OPOI, NOPOI), different levels of desynchronization (0/100/200), and different datasets (ASCADf, ASCADv).

The hyperparameter search was based on the one considered in [30] and adapted to the available resources. Because it showed good results in [30], [45] and [32], and to foster simplicity, random search was selected as the search algorithm. The search space is detailed in section 5.3, as well as all details related to searching and training. The results of each hyperparameter search conducted are presented in the appendix.

The data available was split into a training set, a validation set and an attack set. Both in hyperparameter search and in training, the values of the loss and the accuracy were gathered, both in training and in validation. The hyperparameter selection, as well as the model selection used the validation loss as criteria, as accuracy is usually not considered to be a great side-channel measure [43].

After having gathered competent PTMP, knowledge distillation will be applied. The models that were identified as potentially benefiting from knowledge distillation were the PTMP's attacking the OPOI dataset applied to ASCADv. The OPOI scenario was deemed interesting because it is a task where results were much worse than in the NOPOI scenario, and also a situation where it is likely that a small model works better than a big model. Furthermore, as stated in [34], for classification tasks with a small number of classes or binary detection (two classes) the amount of information transferred from the teacher to the student network is restricted, thus limiting the utility of knowledge distillation. Furthermore, some preliminary experiments conducted in SoS confirmed this statement. Hence, the scope of knowledge distillation was restricted to PTMP in OPOI ASCADv.

In the context of desynchronized traces, two experiments were performed: one in which a dedicated hyperparameter search was conducted specifically for the desynchronized dataset, resulting in the best-performing SoS configurations for this setting; and another in which the best SoS models found for the desync=0 scenario were fine-tuned to adapt to the presence of desynchronization (an additional training session was conducted with this dataset), in a process usually referred to as transfer learning. Experiments were conducted with both levels of desynchronization for the models that were found the most promising.

4.3. Evaluation

After having the trained model, it was evaluated, as in most of the literature, firstly by checking if the correct key converges to being the one the model considers the most likely (successful attack) and with this by investigating how many traces were needed to achieve GE=1. This is done in the following way:

1. From the attacking dataset, 10 attack traces are randomly selected and ordered (an amount which should allow the model to reach GE=1);
2. For each trace, 256 samples are generated - all of which use this trace. For the metadata, all have the correct plaintext, but each has a different key (spanning over all possibilities), and corresponding label;
3. The model is ran in these 256 samples, outputting a Sanity probability for each. Then, these are linearly normalized to achieve a probability distribution over the set of possible keys;
4. This process is done for all attacking traces, therefore achieving a probability distribution over the possible keys for each trace;
5. In (random) order, the cumulative probability distribution over the set of keys is obtained by assuming independence - that is - $P_{k,m} = \Pi_{i=1}^{m} p_{k,i}$, where $p_{k,i}$ is the probability that the model attributed to key k in trace i and $P_{k,m}$ is the cumulative probability attributed to key k considering m traces;
6. From this, if the correct key ever becomes and remains the most likely one, N_{tGE} is computed. If not, the attack is considered failed.

The model is considered to have failed if at least one of the attacks failed, and N_{tGE} is the largest N_{tAR} from the 10 attacks.

Notice, however, that in order to conduct an attack, an attacking dataset with several of samples using the same key is needed. For instance, when attempting to predict the masks, this does not happen (masks are ephemeral). While one could assume knowledge of the masked sbox and compute the evolution of the key as normally, because this is an auxiliary task and metrics are simply used as heuristics and not final results, when training a PTMP, simply the accuracy and average rank are computed.

Furthermore, before conducting an attack, the model was debiased used the mentioned methods, and the results between the raw model and the debiased one were compared.

4.4. Assumptions and Threat Model

In this work, we are assuming an attacker has full knowledge of the AES algorithm taking place, namely assuming knowledge of which intermediate values are leaking (notice only leakage of the first round of AES is analyzed) and of the sbox being used.

Furthermore, during the profiling phase access to a dataset of labeled traces is assumed. These labels are either the plaintext and the key used (in the NTMP scenario), or additionally include the masks applied (for training the PTMP). Usually incorporating knowledge of the masks leads to a a better model, but it is desirable to achieve the best results possible with the fewest assumptions possible.

During the attacking phase, access to traces and corresponding plaintexts is assumed to retrieve the key.

The threat model considered is a white-box model, where the attacker has full knowledge of the implementation and metadata used in training, and grey-box, when masks are assumed to be unknown in training.

5

Experimental Setup

This section intends to detail all specifications about the experiments to ensure maximum reproducibility and to foster a good understanding of the experimentation process.

5.1. Dataset Preparation

As mentioned before, the ASCAD dataset was used for this work, both its variable key and its fixed key version.

Because deep learning is usually sensitive to absolute differences in input [21], each trace $T \in \mathcal{R}^M$ was normalized with a standard normalizer, ensuring the resulting normalized trace T' has mean 0 and unit variance.

Furthermore, an important process in signal processing, particularly in signal analysis for side-channel analysis is feature selection. Typically not all of the points in the dataset are used, and models take advantage in using smaller inputs, balancing the trade-off between optimizing the input size with smaller input and sacrificing the least amount of information possible . For this I followed [30] and considered both OPOI and NOPOI.

The dataset is then split into training, testing, and attacking sets. This split was done as suggested in the original ASCAD paper, with training/attacking split of 50k/10k (fixed key) or 200k/100k (variable key). The training set was then split in 45k/5k and 190k/10k in training/validation, for the ASCADf and ASCADv datasets respectively. The validation set's purpose was to guide the hyperparameter search and the training, while the attacking set only served to evaluate the network after it was fully trained (it is important not to train nor to guide the hyperparameter search according to this dataset, as this would positively skew the results).

Using the best found model, a new training session was run with a new randomized Silly dataset every epoch, and a new dataset every 3 epochs, with each Silly sample being generated with the Silly key value that lead the model to attribute it the highest Sanity probability. Both the training and the validation dataset had Sane and Silly samples in a 1:1 ratio, the attacking dataset had only Sane samples, and they were used to conduct attacks as described in section 4.3.

Furthermore, experiments were run where a desynchronization of 100 / 200 was in place for the NOPOI dataset. For each level of desynchronization, the models r_3, r_{our} and NTMP were experimented with (the model using both masks was left out), both through transfer learning and a dedicated hyperparameter search.

For the scenario where the desynchronization countermeasure is in place, as suggested in [19], data augmentation was employed. Specifically, for each original trace, two additional desynchronized versions were generated: one with a desynchronization value in the range $[0, 50]$ and another in $[50, 100]$ (for desync=100; doubled for desync=200). This not only increases the amount of training data in a balanced way but, more importantly, introduces the notion of shift-invariance to the model — that is, it

implicitly learns that the relevant leakage is invariant to temporal misalignment. This strategy has been empirically shown to be essential when training models to attack desynchronized datasets [19].

5.2. Software and Hardware

My scripts were ran on Deflt Blue, in the gpu-a100 partition. The jobs ran on 2 NVIDIA A100 Tensor Core GPU with 80 GB of video RAM each. Some scripts also ran in an online GPU cluster with similar characteristics. My code was written in Python 3.8.12, and the main deep learning library used was Tensorflow 2.4.1 [1].

5.3. Training and Evaluation Pipeline Details

As mentioned in the Methodology section, the pipeline in the model development is mainly composed of 4 stages:

- PTMP hyperparameter search;
- PTMP training and evaluating;
- SoS hyperparameter search;
- SoS training.

Each of these stages has important details to go over as well as important results to be presented, for each dataset. In this section the details about each stage will be described, and the results will be left for section 6.

5.3.1. PTMP Hyperparameter Search

Although there are some differences between the networks used for ASCAD fixed key and for ASCAD variable key, the architectures are similar - they are composed of convolution layers followed by dense layers. In both cases, the convolution layers are followed by pooling layers and by batch normalization. After the dense layers searched for, an extra dense layer with 256 neurons with softmax is applied to output a probability distribution over the set of possible masks. The activation functions considered were `relu` and `selu`, searched for all layers.

Furthermore, in both cases, search was conducted over the optimizer to be employed (between Adam and rmsprop), and the learning rate (log-Uniformly). In the fixed key scenario the learning rate was fixed during training, this was changed in the variable key setting, where an exponential decay in the learning rate was employed. The decay factor was searched log-Uniformly in $[0.9, 0.99]$ for each epoch.

Hyperparameter search lasted 200 trials, each with 10 epochs, and with an early stop if no improvement was witnessed after 3 epochs (validation accuracy < $\frac{1}{256}$).

The complete description of the search space for the mask preedictors for both datasets is provided in table 5.1.

Parameter	ASCADf	ASCADv
`Convolution Layers`	{1,2,3}	{2,3,4}
`Filters`	$\{8, 16, 32, 64, 128\}$	$\{32, 64, 128\}$
`Kernel Size`	$\{10, 15, 20, 25, 30\}$	$\{10, 20, 30\}$
`Stride`	$\{5, 10, 15\}$	$\{5, 10, 15\}$
`Pooling type`	{`max, average`}	{`max, average`}
`Dense Layers`	$\{0, 1, 2, 3\}$	$\{1, 2, 3\}$
`Neurons (Dense)`	$\{50, 100, 256\}$	$\{64, 96, 128\}$
`Activation functions`	{ `relu, selu` }	{ `relu, selu` }
`Optimizer`	{ `adam, rmsprop` }	{ `adam, rmsprop` }
`Batch Size`	$\{32\}$	$\{16, 32, 64\}$
`Learning Rate`	{0.0001, 0.001, 0.01}	[0.000001, 0.0001] $\times$batch size
`Decay rate`	1	[0.9, 0.99]

Table 5.1: Comparison of architecture-related hyperparameters.

Some differences between the two scenarios were motivated by one dataset being bigger and harder than the other. This was the case, for example, for the increased number of convolution layers, to further reduce the dimension. Other changes were simply motivated by observation - because I worked with the fixed key dataset first, I noticed, for example, that the number of filter per layer being chosen was always one of the top values, leading me to exclude the smaller values in future searches, to reduce the search complexity. Hence, I refined my approach as I worked on this problem. But because the results were satisfying for ASCADf, I did not find the need to re-do them. The batch size value is different because of memory restrictions.

5.3.2. PTMP Training and Evaluating

After selecting the model that performed best in terms of validation loss from the hyperparameter search, the model is then trained for 200 epochs with the found optimizer and loss function categorical cross-entropy. To evaluate this network, we gather measures of accuracy and mean rank.

5.3.3. Knowledge Distillation

The knowledge distillation process was also subject to hyperparameter search. In particular, the searchable aspects of the knowledge distillation were the temperature ($T \in [1, 5]$), the alpha ($\alpha \in [0.1, 0.9]$) and some aspects of the model's architecture. Knowledge distillation was applied to both models in ASCADv OPOI PTMP. All parameters were searched uniformly in their search spaces.

The complete description of the search space for the knowledge distillation process is provided in table 5.2.

Each trial trained the student for up to 10 epochs with the Adam optimizer, minimizing the combined distillation loss

$$\mathcal{L} \;=\; \alpha\, T^2\, \mathrm{CE}\big(\sigma(s/T),\, \sigma(t/T)\big) \;+\; (1-\alpha)\, \mathrm{CE}\big(\sigma(s),\, y\big), \tag{5.1}$$

where t are the teacher probabilities, s the student logits, and σ the softmax function. Early stopping was triggered if validation accuracy failed to exceed preset thresholds at epochs 2, 4, or 6, set empirically to $\frac{1}{256}, \frac{2}{256}, \frac{3}{256}$. The student achieving the lowest validation loss was retained and its weights saved for final evaluation.

The last dense layers always has shape 256 to get the probability distribution over the masks. Searching lasted for 50 trials, and the final model was trained for 100 epochs.

Parameter	PTMP ASCADv OPOI
Batch size	{ 32, 64, 128}
Initial LR	[0.00001, 0.0001]
Convolution layers:	{1,2}
Number of kernels:	{32,48,64}
Kernel size:	{10,20,30}
Number of dense layers:	{1,2}
Number of neurons:	{128,256}
Temperature:	[1,5]
Alpha:	[0.1,0.9]

Table 5.2: Searched hyperparameters for Knowledge Distillation.

5.3.4. SoS Hyperparameter Search

Once again , whilst the baseline structure of SoS for ASCADf and ASCADv is similar, there is some variability to the approach taken in each dataset, explained firstly by the nature of the task, and secondly due to the knowledge that the experiments done in ASCADf gave me, making the search in ASCADv more informed. Furthermore, as the task of attacking ASCADv is much harder than attacking ASCADf, some parameters relating to the architecture of SoS were searched for, to ensure the model is optimized to the task, as opposed to the ASCADf case, where only training hyperparameters were searched for. A different hyperparameter search was conducted for each pair of SoS version and Silliness generation method. The detailed architecture of SoS can be found in the appendix, for organization purposes it is not detailed here.

The complete description of the search space for SoS is provided in table 5.3.

Parameter	ASCADf	ASCADv
Batch size	{ 32, 64, 128}	{16, 32}
Initial LR	{0.0001, 0.001, 0.01}	[0.0000005, 0.00005] $\times$ batch size
Decay Rate	[0.9, 0.99]	[0.9, 0.99]
Loss Function:	{hinge, binary crossentropy}	{hinge, binary crossentropy}
Dense layers:	4	[4, 6, 8, 10]
Embedding size:	32	[12, 24, 36]
Output size:	100	[50, 100, 200]

Table 5.3: Comparison of architecture-related hyperparameters.

The number of epochs per trial and number of trials was different in the 2 datasets, due to hardware limitations. In the ASCAD fixed dataset, a more thorough search of 100 trials and 20 epochs per trial was conducted. This was, however, not possible to run within the 24 hour time limit for the ASCAD variable dataset, and instead, the number of epochs per trial was set to 15, with early dropouts in epochs 3, 6, 9 and 12 (to combat the time limit imposed by the cluster and the complexity caused by the large dataset and the hyperparameter search), in case the validation accuracy was below thresholds 0.5, 0.55, 0.6 and 0.65 respectively. These thresholds were set empirically. The maximum number of trials that ran during the 24 hours of running time I had was between 10 (hard silliness generation) and 40.

An important note is that the hinge loss function expects outputs between -1 and 1, which needed some

slight adaptation of the output of the model, which by default is bounded by 0 and 1. After witnessing binary cross-entropy out performing the hinge loss function consistently, and because of the more convoluted approach it needed, eventually only the binary crossentropy function was considered.

5.3.5. SoS Training and Evaluating

After selecting the model that performed best in terms of validation loss from the hyperparameter search, the model is then trained for 200 epochs with the Adam optimizer and the found loss function, and found decay rate each epoch. With the best trained model, the attacking set was attacked as described in section 4.3, and N_{tGE} was computed.

One challenge that was found when working with ASCADv was that hyperparameter search trained the network for a maximum of 15 epochs. This network was then intended to be trained for 200 epochs, but because the learning rate and the decay rate were chosen specifically for a short training session, these were not optimal for a longer one, and the model usually showed no improvement past epoch 50. To address this, a cyclical learning rate schedule was employed [39]. This reset the learning rate every 15 epochs to 0.95 of its value 15 epochs back, ensuring more longevity of the improvement of the model. A generic evolution of the learning rate when cyclical learning rate is in place is plotted in figure 5.1.

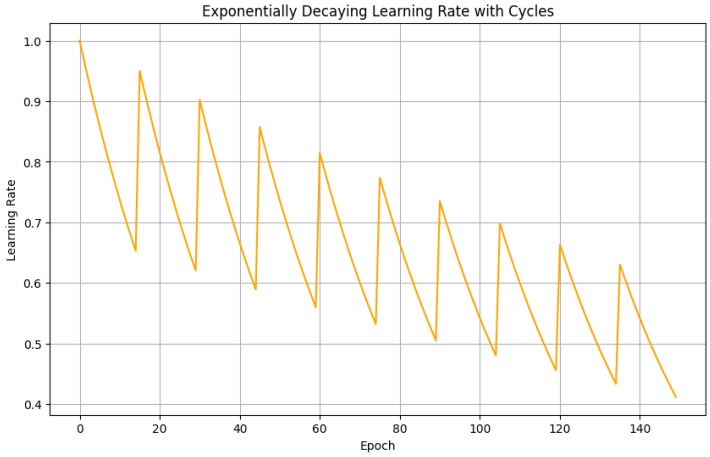


Figure 5.1: Example of cyclical learning rate evolution starting at 1 with decay rate 0.98 and resetting to 0.95 of its value 15 epochs back.

5.3.6. NTMP

As mentioned before, one of the experiments conducted was forfeiting the need of pre-training a part of the model for predicting the masks. In this experiment, PTMP was replaced by NTMP, with similar structure but no previous training. Similarly to the other cases, the structure for this block was searched for, with certain simplifications due to limited resources. Namely, all the convolution layers have the same number of filters and the same kernel size. Also, all the dense layers have the same number of neurons. The activation functions are all relu and the pooling layers are all average. Furthermore, one different aspect between this scenario and the other ones, is that the output of NTMP does not have shape 256. Instead, it has the same shape as the embeddings. Hence, the model is implicitly learning not only to predict the mask (hopefully) [25], but also how to represent this information in the best way. Apart from this, its structure is equal and search space similar to previous scenarios:

The complete description of the search space for the NTMP block of SoS is provided in table 5.4.

Parameter	ASCADf/ASCADv
`Convolution Layers`	{2,3}
`Filters`	$\{32, 64, 128\}$
`Kernel Size`	$\{10, 20, 30\}$
`Dense Layers`	$\{1, 2, 3, 4\}$
`Neurons (Dense)`	$\{32, 64, 128\}$

Table 5.4: NTMP search space

6

Experimental Results

In this section the results of the most relevant experiments conducted are presented. Its main purpose is not to discuss them and analyze them, but simply to display them, as their discussion is left for the next chapter.

6.1. Mask Predictors

Firstly, the results of the Pre-trained Mask Predictors will be presented. These correspond to the models eventually incorporated in the corresponding version of SoS, that has its results presented in the following section. As mentioned before, whilst when attacking the key, one possesses a dataset with an attacking set always using the same key, that is not true for when attacking a mask, which is always variable. It is relevant to notice that there is a way to work around this: one can assume the masked sbox output is known to the attacker, and with that and the predicted mask, reconstruct the sbox output and attack the key, computing N_{tGE}. Because this is a relatively easy task, simply the accuracy on single traces is presented. In table 6.1 and 6.2, the results of the mask predictors acting on ASCADf and ASCADv (respectively) are presented. Table 6.2 has an additional row, corresponding to the results of Knowledge Distillation. Furthermore, in table 6.2. in the desynchronization rows, a comparison is made between naive training and training with data augmentation.

Parameter	r_3	r_{out}
`NOPOI`	0.843 (1.31)	0.844 (1.30)
`NOPOI desync100`	0.156 (7.50)	0.1335 (8.25)
`NOPOI desync200`	0.1462 (7.98)	0.0961(9.51)
`OPOI`	0.0266 (45.20)	0.0195 (61.94)

Table 6.1: Accuracy (average rank) of best networks predicting masks in ASCADf

6.2. SoS

Here the best results obtained for each dataset are presented. The values of N_{tGE} are presented, and X denotes an unsuccessful attack. In table 6.3 the results of the 4 SoS versions attacking ASCADf are presented. Tables 6.4 through 6.7 refer to the various experiments conducted with ASCADv : Table 6.4 presents the results of the 4 base models on NOPOI/OPOI, while the next one shows the results of those same models after debiasing. The last 2 tables have the results of SoS attacking the ASCADv dataset with desynchronization : table 6.6 refers to the scenario where a dedicated hyperparameter search was conducted, while Table 6.7 displays the results of the experiments with transfer learning from the original models to the dataset with desynchronization.

Parameter	r_3	r_{out}
NOPOI	0.998 (1.0)	0.990 (1.0)
NOPOI desync100	0.8355 [0.8245] (1.2)	0.8381 [0.8039] (1.2)
NOPOI desync200	0.9527 [0.8421] (1.1)	0.775 [0.6639] (1.3)
OPOI	0.0248 (46.9)	0.0127 (78.8)
OPOI student	0.0281 (42.3)	0.0166 (64.9)

Table 6.2: Accuracy (average rank) of best networks predicting masks in ASCADv. In [ ] are the results without data augmentation

Parameter	r_3	r_{out}
NOPOI	1	1
NOPOI desync100	1	1
NOPOI desync200	1	1
OPOI	1	1

Table 6.3: N_{tGE} of SoS in ASCADf

Parameter	r_3	r_{out}	both	NTMP
NOPOI	4	3	3	X
OPOI	X	X	X	X

Table 6.4: N_{tGE} of raw SoS in ASCADv

Parameter	$r_{out} + r_3$	both (prior)	ntmp (prior)
NOPOI	1	16	1 (ch)
OPOI	X	X	X

Table 6.5: N_{tGE} of the debiased SoS in ASCADv

Parameter	r_3	r_{out}	NTMP
NOPOI desync100	X	X	X
NOPOI desync200	X	X	X

Table 6.6: N_{tGE} of SoS in ASCADv desync - dedicated search

Parameter	r_3	r_{out}	NTMP
NOPOI desync100	$\sim$ 900 (debiased)	X	X
NOPOI desync200	X	X	X

Table 6.7: N_{tGE} of SoS in ASCADv desync - transfer learning

7

Discussion and Future Work

This chapter reflects on the key findings and limitations of the work presented in this thesis, drawing connections to broader trends in side-channel analysis and deep learning. It examines the implications of architecture choices, data preprocessing, data generation, and countermeasures, highlighting how these factors influence performance. Additionally, the chapter outlines several promising directions for future work that aim to further enhance attack robustness and efficiency.

7.1. Mask Predictors

Regarding mask predictors, a significant decline in performance is seen when going from NOPOI to OPOI, in both datasets. This is in line with the state-of-the-art, where N_{tGE} goes from 1 to 78/87 in ASCADv/ASCADf [30]. By examining the SNR plots presented in [2], this can be interpreted by the fact that, even though the selected interval contains information on the two pairs of $(sbox(p_3 \oplus k_3) \oplus r_3, r_3)$ and $(sbox(p_3 \oplus k_3) \oplus r_{out}, r_{out})$, there are significant spikes on the SNR of the masks outside this interval that are not considered, especially for r_{out}. This mask was also the one with worst performance in OPOI - whilst having similar performance in NOPOI. In addition, subsampling brings the advantage of shortening the input size, making the full trace not overwhelmingly big, and therefore ”mimicking” the advantage of picking OPOI, without sacrificing as much information. Nevertheless, this work shows that the mask itself is hard to predict in OPOI, and explains the hardness of attacking it. It could be interesting to investigate if the masked sbox value $sbox(p_3 \oplus k_3) \oplus r_3$ is also significantly harder to predict in OPOI compared to NOPOI, or if, in contrast, the hardness of attacking OPOI comes solely from the hardness of predicting the mask. This should be relatively straightforward, but falls outside the scope of this thesis, so it is left for future work.

Results achieved for ASCADf were significantly worse than the ones attained in ASCADv. Although this, at first, seems counterintuitive, as ASCADv should be a harder dataset than ASCADf, there are clear explanations for this: firstly, the fact that the key is variable for each trace does not directly make predicting the mask harder - the masks are variable in both scenarios. Secondly, as ASCADv is a larger dataset, and experiments were ran for a longer time, it is expected that results will be better. Thirdly, as my experiments began with ASCADf and then evolved for ASCADv, the search space and overall training process were more refined when I was working with ASCADv, which also explains part of the improvement. I believe if more work was devoted to improving the mask predictors in ASCADf, there was room for improvement. However, as the results achieved by SoS were already optimal, this was not considered to be a top priority within this project.

One important remark about mask predictors is the fact that while desynchronization caused a decline in accuracy and average rank, the resulting models were still quite competent. In fact, one surprising result was that the model guessing mask r_3 actually improved in performance when faced with a desynchronization of 200, compared to 100, both with and without data augmentation. Sanity checks were performed and the experiments were repeated, but this kept happening. Although counterintuitive (the task of desync=200 is harder than desync=100), the model might be benefiting from training on higher

desynchronization, making it more robust. Also, the fact that this happened with mask r_3 and not mask r_{out} is unexpected, as a priori no significant differences in the nature of their leakage was considered. This might be because of the nature of the leakage of mask r_3, making desynchronization not a good countermeasure to hide it. Furthermore, it is important to note that indeed data augmentation proved to be a useful step to enhance performance, and that, while desynchronization made performance worst, this was only slightly, and results were still good both for guessing r_3 and r_{out}.

Finally, regarding knowledge distillation, a marginal improvement was observed when comparing the student model to its teacher counterpart. This aligns with findings in the literature, where KD typically yields moderate gains, particularly because the effectiveness of the student model is inherently limited by the performance of the teacher [14] [11]. Nevertheless, KD remains a useful tool, especially in scenarios where model compression or generalization is desired [34].

The experiments presented in this thesis serve as a proof of concept. It may therefore be worthwhile to further explore knowledge distillation in the context of standard label classifiers, rather than models trained to predict the mask values.

Another important remark is the high variance in model performance that was observed during hyperparameter search, both for PTMP and across SoS models. For the same task, different iterations even with similar hyperparameters had significantly different results. Hence, even though the different architectures that resulted from hyperparameter search in two similar tasks (for example, predicting r_3 and r_{out}) were sometimes extremely different, and this difference does seem arbitrary, the results observed during hyperparameter search indicated that indeed these differences were important and that different leakage entails a different optimal model. Of course, the models found are not necessarily optimal, and the differences in performance could be simply derived from the random initialization of weights, but nevertheless, it is an indicator that this task is extremely sensitive to model architecture.

7.2. Sane or Silly

Firstly, it is noticeable that the results when using the ASCADf dataset were overwhelmingly good, in spite of the desynchronization considered and feature selection method chosen. This, however, is easily explained when examining the nature of the dataset and the architecture of the model.

The key is fixed throughout the training process, the model receives the plaintext as input, as it is looking to predict $sbox(p_3 \oplus k_3)$. Because the model has access to p_3 via input, and $sbox()$ and k_3 are fixed throughout the training process, all that the model really needs to do to perform highly is to learn the simple function $f(x) = sbox(k_3 \oplus x)$ during the training process and apply them to input p_3. With effect, after the model is trained, its efficacy is independent of the trace fed to it - it can discern a sane combination of label and plaintext from a silly one without considering the trace. This also explains why the model is not affected by varying performances of PTMP, as this prediction also becomes independent of the mask used.

Interestingly, although performance is stable across all experiments, the model's confidence varies with the desynchronization chosen and with the feature selection method. This indicates that the model is using the trace in some parts of the training process, or at least is being confused by the PTMP undecided-ness.

Considering ASCADv, the results are quite satisfying: SoS was able to attack ASCADv with only 1 trace, both when mask information was assumed to be available during training, but also with no assumption of knowledge of masks in the profiling phase.

Furthermore, it is interesting to note that mixing information from both masks seems to be more effective when done in different models, rather than in one. There are two main reasons for this: firstly, the information of 1 mask only suffices to retrieve the key, so extra information can lead to unnecessary complexity and confusion. However, the main reason attributed to this phenomenon is that combining different models is inherently better. Models appeared to have an intrinsic and independent bias, which was the main problem with the models. Combining models after training will cancel the biases, solving 2 problems at once, and arriving at a better model.

Because there is much information leaking that may be useful to retrieve the key, I believe this is also

something fairly unexplored in the literature - building separate models to attack the key by exploiting different leaky information, and combining the information extracted with them in different ways.

When experimenting with different data generation methods, it was observed that generating hard samples through hard negative mining never enhanced performance. There is a clear trade-off here, between larger training times and a better dataset. Because the ASCAD dataset is quite big, raising the training time was never worth it. Changing the dataset each epoch was a crucial improvement for the NTMP SoS. This model always showed a different learning curve from others, as shown in figure 7.1: while other (r_3 and r_{out}) would often reach peak performance early on in training (sometimes even during hyperparameter search), the NTMP model always improved in a less smooth manner, achieving peak performance later on in training - probably due to it having the additional task of learning the masks and learning a good way to represent them. Perhaps because of this added difficulty, the static dataset version was unable to attack ASCAD, even when debiased. Only in the changing silliness version, with prior debiasing, was it able to attack ASCAD, and only requiring 1 trace. Hence, it was observed that while ASCAD was a big dataset that did not usually need extra diversity in it, changing the silly samples every epoch was crucial for this particular model to achieve good performance.

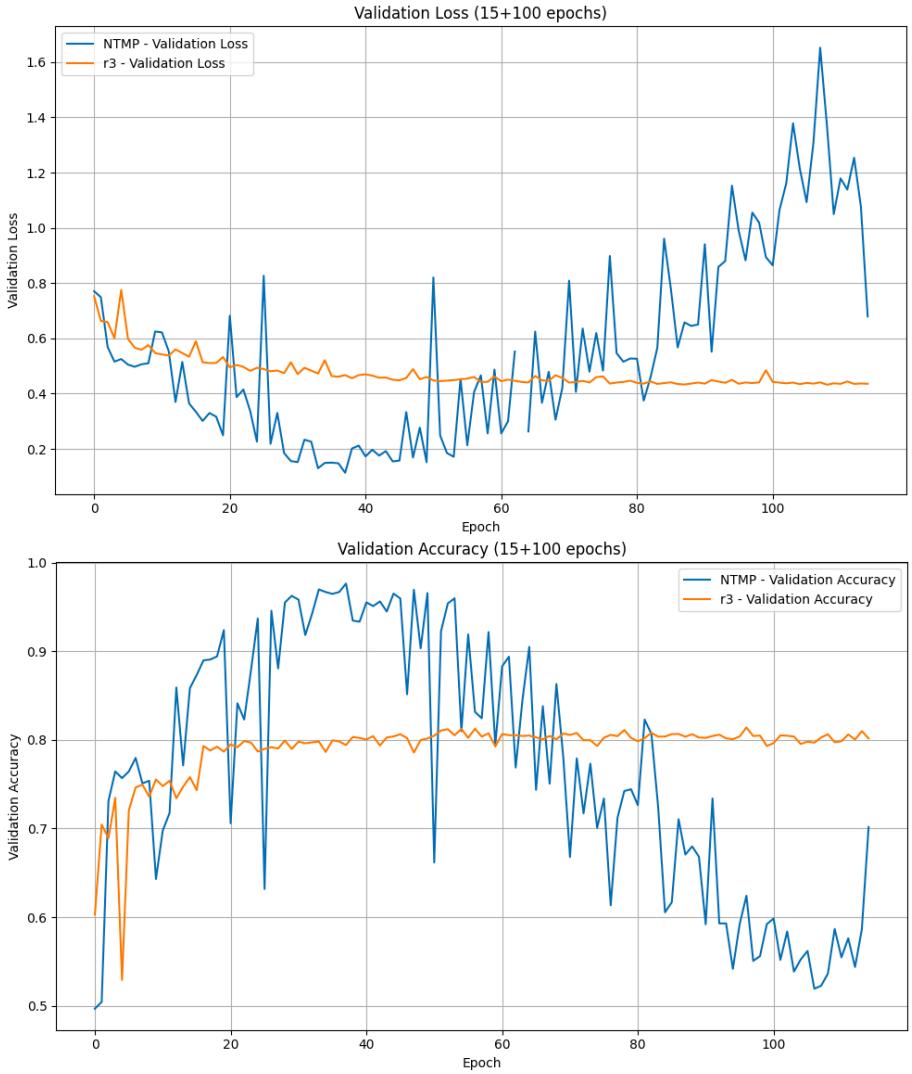


Figure 7.1: Evaluation of validation loss and accuracy in the training process of models r_3 and NTMP

Left for future work is experimenting with other data generation methods. In particular, these generation methods were heuristically selected and not searched for. Hence, there is room for improvement here, namely by searching the frequency of new silliness generation (here 3 epochs, chosen heuristically) or by implementing evolutionary algorithms. Furthermore, it may also be beneficial to tweak the ratios of

Sane and Silly samples. Due to time and memory constraints, a ratio of 1 was maintained. This, again, may be subject to search.

Regarding the debiasing method used, important conclusions may be drawn. Firstly, the prior based method was the single-model debiasing method that performed the best, and was the only one used in practice. In the best-performing models — those able to attack ASCADv with very few traces — debiasing made the model perform the same or slightly worse. However, in the less effective models, which required dozens or hundreds of traces to attack ASCADv, debiasing had quite a positive effect. In fact, prior-based debiasing made the NTMP model go from not being able to attack ASCADv to attacking it in 1 trace. This suggests that learning to remove bias is part of the training process itself, and a challenge independent from the rest of the learning process. Models that had good raw performance (no debias) rarely benefited from prior-based debiasing (like r_3, r_{out} and both). Instead, when their information is combined, the bias is diluted due to their independence, attacking ASCADv in 1 trace only. The combination method used to combine the r_3 and r_{out} models was loss-weighted averaging.

Finally, it is worth noting that although desynchronization significantly increased the difficulty of attacking the dataset, SoS was still able to successfully attack ASCADv at a desynchronization level of 100. This was only possible when the data augmentation process described in [19] was applied. Additionally, model debiasing was necessary; two debiasing methods were tested and yielded similar results. Besides the standard prior-based method, another effective approach involved combining knowledge from two SoS iterations: one using the mask model trained with desync=100 and another with desync=200 - both trained on the dataset with desync=100 (based on the assumption that a model robust against desync=200 should also perform well against desync=100). Although both models performed similarly, neither could conduct a successful attack on their own. It is important to mention that when N_{tGE} is large, this figure fluctuates depending on the attacking set chosen. Because the attacking set is quite large and the attacking traces are randomly chosen from this set in a random order, this figure was kept as an estimate of 900.

A similar strategy was attempted for attacking the dataset with desync=200 -both using the prior method and combining iterations with mask models trained on desync=200 and desync=400. However, these attempts were unsuccessful. Overcoming this countermeasure more effectively, as well as experimenting with higher desynchronization levels and additional countermeasures such as noise injection, shuffling, and multiplicative masking, are left as directions for future work.

In the OPOI scenario, none of the attacks were effective, which aligns with the poor performance of mask predictors. Literature also supports the idea that smaller models tend to perform better under OPOI conditions [47, 32]. Thus, while testing the SoS approach in this setting was valuable, the architecture likely exceeded the optimal model complexity for this scenario, and this approach seems too involved.

8

Conclusion

To conclude this work, the initial research questions are revisited to assess the progress made in addressing them, as well as the broader impact on the field of side-channel analysis.

The most important contribution of this thesis is that it demonstrates that language models can be effectively applied in an SCA setting. The proposed framework, Sane or Silly, achieved optimal results when attacking both the ASCADf and ASCADv datasets, successfully retrieving the key using only a single trace. Moreover, this work addresses the main future work proposed in [20], eliminating the need for a pre-trained mask predictor and thereby relaxing the profiling assumptions required to attack ASCADf and ASCADv.

Desynchronization proved to be a somewhat strong countermeasure, changing the amount of traces needed to conduct an attack from 1 to 900 with a desynchronization level of 100, and completely avoiding exploits with desync = 200. This is another achievement and novelty of this work.

Interestingly, while desynchronization made it harder to retrieve the key, masks proved to be relatively easy to recover under desynchronization. This might be attributed either to the leakage model or to model-specific vulnerabilities.

Other important insights obtained from this project arise from the separation of mask and key prediction. This separation revealed that predicting masks under OPOI is challenging, likely due to useful SNR peaks lying outside the selected range. This hardness, and perhaps hardness in predicting the remaining values, prevented SoS from attacking the OPOI dataset.

Different architectures were explored depending on the dataset characteristics, with final models ranging between 100k and 2 million trainable parameters. Although an extensive hyperparameter search might initially seem excessive, results showed that SoS is highly sensitive to hyperparameter tuning.

In summary, this thesis proposes an alternative framework for profiling SCA, demonstrates its efficacy on ASCADv even against desynchronization, and removes the dependency on secret mask knowledge during profiling. These results open new doors for further exploration of language-model-based methods in side-channel analysis.

Specifically, future work could consider:

1. Investigating whether or not the OPOI hardness comes solely from mask prediction;
2. Conducting Knowledge Distillation to a standard classifier;
3. Applying attention mechanisms to the input embeddings;
4. Exploring alternative Silly sample generation methods;
5. Exploring alternative hyperparameter seach methods;
6. Designing new architectures with a higher resilience to desynchronization;
7. Attacking different key bytes with SoS with a similar architecture;
8. Attacking the ASCADv2 dataset, with different countermeasures in place (shuffling and multiplicative masking).

This approach does not seem to be particularly well-suited for OPOI, as no attack in this work was successful, and the literature suggests simpler models. Furthermore, because using them assumes mask knowledge during training, improving the performance of SoS with this point selection method is not considered a priority.

Knowledge distillation showed promising results but is yet to be tested on a standard classifier. This is unexplored in the literature and could lead to interesting results, particularly in the development of simpler models. Hence, this is identified as one of the most promising aspects for future work.

It is important to test SoS against other bytes of AES. From a practical standpoint, this would confirm the idea that SoS can not only retrieve the second byte of the key but the entire key. According to the literature, this should be a challenge of similar difficulty to retrieving the second byte of the key [7]. However, as this work confirmed, different leaky values with different leakage patterns can result in very different tasks for the model to learn, so I believe this task should be a priority. Also, even though attacking ASCADv2 with all countermeasures seems extremely difficult (only accomplished once in the literature, to the best of my knowledge), experimenting with SoS on it (perhaps without some of its countermeasures) could potentially yield valuable insights.

The rest of the future work consists of guidelines for possible improvements to SoS. These are the ones I would attempt if I had more time to work on it. There are certainly other aspects that could be improved, but these serve as recommendations for anyone wishing to further improve SoS.

From a bird's-eye view, this work showed that an attacker could exploit electromagnetic emanations to attack an AES device with the help of a language model. Success is demonstrated to be possible if the target behaves like an ATMega8515 (used in the ASCAD databases) and if the attacker has access to a cloned copy of the device. Desynchronization under 100 time ticks—about 50 nanoseconds—proved to be an insufficient countermeasure. This finding opens the door to new architectures that could pose practical threats to AES.

9

Reflection Note and Acknowledgments

I started working on this project in November and submitted it in the end of July, 9 months of work later. As this was the biggest project I had ever worked on, and because it was a big part of my life for a long time, I felt the need to include a section to discuss my experience with it, as well as to record some aspects of the process that would otherwise be left undocumented.

The topic for my thesis was chosen by me, as it combined two of the research fields that interested me the most: cryptography and machine learning. In light of this, I approached Stjepan Picek in late September to learn more about possible theses in this area. I am grateful that he took me under his guidance in this project, and I extend to him my sincere appreciation. Before starting this project, my background in SCA was nonexistent.

I must also thank Jakob Söhl for being my TU Delft supervisor for this project, and for always making sure I was up to date with formalities and being ready with advice.

While I am a decent coder and have been coding for some years now, my background is not that of a software engineer, and I sometimes definitely lacked organization and efficiency. This was, for me, an opportunity to improve my software skills, and thanks to the vast resources available online, there was nothing I intended to do that was left undone due to lack of coding skills.

The most important bottleneck that hindered my progress was DelftBlue's capacity. The cluster worked with Slurm, meaning one would run a program by submitting a request, and the requests were placed in a queue, giving priority to the least demanding ones. The scripts I ran required requesting the maximum resources—hyperparameter search meant I requested the maximum time (24 hours) to search as widely as possible, and the size of the dataset meant I requested the maximum memory. (While I don't discard the possibility that there was room for optimization in my scripts, the nature of the programs I intended to run was heavy). This meant waiting sometimes a full week to get results. Hence, while at first I kept on reading, writing, and coding while waiting for the scripts to run, towards the end this became a serious problem, as all that was left was getting some results, leading me to also run some scripts on a paid online GPU to accelerate the process. While I believe the resources available were insufficient, I am nevertheless grateful for having had the opportunity to use DelftBlue.

Apart from SCA and language models, this thesis taught me a lot about tools such as Slurm and Git, as well as about how to conduct a research project — especially in terms of organization and prioritizing long-term success.

I am happy not only with the learning process this thesis granted me over the last months, but also with the results, which are consistent with State of the Art and expand our knowledge on this field.

At last, I owe a word of gratitude to my family, with whom I lived while working on my thesis for the last 9 months. This helped making this period not only quite productive but also peaceful and happy.

References

[1] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC]. URL: https://arxiv.org/abs/1605.08695.

[2] Ryad Benadjila et al. "Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database". In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 53. URL: https://eprint.iacr.org/2018/053.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] Olivier Bronchain, Gaëtan Cassiers, and François-Xavier Standaert. *Give Me 5 Minutes: Attacking ASCAD with a Single Side-Channel Trace*. Cryptology ePrint Archive, Paper 2021/817. 2021. URL: https://eprint.iacr.org/2021/817.

[5] Olivier Bronchain et al. "Leakage Certification Revisited: Bounding Model Errors in Side-Channel Security Evaluations". In: *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I*. Santa Barbara, CA, USA: Springer-Verlag, 2019, pp. 713–737. ISBN: 978-3-030-26947-0. DOI: 10.1007/978-3-030-26948-7_25. URL: https://doi.org/10.1007/978-3-030-26948-7_25.

[6] George Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989). Contains the original proof of the Universal Approximation Theorem., pp. 303–314. DOI: 10.1007/BF02551274.

[7] Maximilian Egger et al. "A Second Look at the ASCAD Databases". In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by Josep Balasch and Colin O'Flynn. Cham: Springer International Publishing, 2022, pp. 75–99. ISBN: 978-3-030-99766-3.

[8] Zibo Gao et al. *I Know What You Said: Unveiling Hardware Cache Side-Channels in Local Large Language Model Inference*. 2025. arXiv: 2505.06738 [cs.CR]. URL: https://arxiv.org/abs/2505.06738.

[9] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. "Templates vs. Stochastic Methods". In: *Cryptographic Hardware and Embedded Systems - CHES 2006*. Ed. by Louis Goubin and Mitsuru Matsui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 15–29. ISBN: 978-3-540-46561-4.

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

[11] Jiuxiang Gou et al. "Knowledge Distillation: A Survey". In: *International Journal of Computer Vision* 129 (2021), pp. 1789–1819.

[12] Chirag Gupta and Aaditya Ramdas. *Top-label calibration and multiclass-to-binary reductions*. 2022. arXiv: 2107.08353 [cs.LG]. URL: https://arxiv.org/abs/2107.08353.

[13] Joshua Harrison, Ehsan Toreini, and Maryam Mehrnezhad. "A practical deep learning-based acoustic side channel attack on keyboards". In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2023, pp. 270–280.

[14] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: *NIPS Deep Learning and Representation Learning Workshop*. 2015. URL: https://arxiv.org/abs/1503.02531.

[15] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 2nd ed. CRC Press, 2014.

[16] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017. arXiv: 1609.04836 [cs.LG]. URL: https://arxiv.org/abs/1609.04836.

[17] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology — CRYPTO’ 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9.

[18] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology — CRYPTO ’96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[19] Marina Krček et al. “Shift-Invariance Robustness of Convolutional Neural Networks in Side-Channel Analysis”. In: *Mathematics* 12.20 (2024). ISSN: 2227-7390. DOI: 10.3390/math12203279. URL: https://www.mdpi.com/2227-7390/12/20/3279.

[20] Praveen Kulkarni et al. *Order vs. Chaos: A Language Model Approach for Side-channel Attacks*. Cryptology ePrint Archive, Paper 2023/1615. 2023. URL: https://eprint.iacr.org/2023/1615.

[21] Yann LeCun et al. “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade*. Ed. by Gary Orr and Klaus-Robert Müller. Springer, 1998, pp. 9–50.

[22] Jean-Marie Lemercier et al. *An Independence-promoting Loss for Music Generation with Language Models*. 2024. arXiv: 2406.02315 [cs.SD]. URL: https://arxiv.org/abs/2406.02315.

[23] Shuang Li et al. “Pre-Trained Language Models for Interactive Decision-Making”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 31199–31212. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/ca3b1f24fc0238edf5ed1ad226b9d655-Paper-Conference.pdf.

[24] Ali Madani et al. *ProGen: Language Modeling for Protein Generation*. 2020. arXiv: 2004.03497 [q-bio.BM]. URL: https://arxiv.org/abs/2004.03497.

[25] Loïc Masure et al. *Don’t Learn What You Already Know: Scheme-Aware Modeling for Profiling Side-Channel Analysis against Masking*. Cryptology ePrint Archive, Paper 2022/493. 2022. URL: https://eprint.iacr.org/2022/493.

[26] Math Stack Exchange. *A name for layered directed graph as in a fully connected neural network*. https://math.stackexchange.com/questions/2048722/a-name-for-layered-directed-graph-as-in-a-fully-connected-neural-network. Accessed: April 4, 2025. 2025.

[27] Aditya Krishna Menon and Robert C. Williamson. “On the Deterministic Relationship Between Class Balance and Predictive Performance”. In: *International Conference on Machine Learning (ICML)*. 2019.

[28] National Institute of Standards and Technology (NIST). *Advanced Encryption Standard (AES)*. FIPS Publication 197. See §5 for the brute-force key-search complexity discussion. U.S. Department of Commerce, National Institute of Standards and Technology, Nov. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[29] Yuqi Nie et al. *A Time Series is Worth 64 Words: Long-term Forecasting with Transformers*. 2023. arXiv: 2211.14730 [cs.LG]. URL: https://arxiv.org/abs/2211.14730.

[30] Guilherme Perin, Lichao Wu, and Stjepan Picek. *Exploring Feature Selection Scenarios for Deep Learning-based Side-Channel Analysis*. Cryptology ePrint Archive, Paper 2021/1414. 2021. URL: https://eprint.iacr.org/2021/1414.

[31] Stjepan Picek et al. *SoK: Deep Learning-based Physical Side-channel Analysis*. Cryptology ePrint Archive, Paper 2021/1092. 2021. URL: https://eprint.iacr.org/2021/1092.

[32] Jorai Rijsdijk et al. *Reinforcement Learning for Hyperparameter Tuning in Deep Learning-based Side-channel Analysis*. Cryptology ePrint Archive, Paper 2021/071. 2021. DOI: 10.46586/tches.v2021.i3.677-707. URL: https://eprint.iacr.org/2021/071.

[33] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[34] Ahmad Sajedi and Konstantinos N. Plataniotis. “On the Efficiency of Subclass Knowledge Distillation in Classification Tasks”. In: *arXiv preprint arXiv:2109.05587* (2021). URL: https://arxiv.org/abs/2109.05587.

[35] Tim Salimans and Durk P Kingma. “Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/ed265bc903a5a097f61d3ec064d96d2e-Paper.pdf.

[36] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176.

[37] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. “Training Region-based Object Detectors with Online Hard Example Mining”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 761–769. DOI: 10.1109/CVPR.2016.91.

[38] Michael Sipser. *Introduction to the Theory of Computation*. 3rd. Cengage Learning, 2012. ISBN: 978-0534950972.

[39] Leslie N. Smith. “Cyclical Learning Rates for Training Neural Networks”. In: *arXiv preprint arXiv: 1506.01186* (2015). Available at https://arxiv.org/abs/1506.01186.

[40] F.-X. Standaert, E. Peeters, and J.-J. Quisquater. “On the masking countermeasure and higher-order power analysis attacks”. In: *International Conference on Information Technology: Coding and Computing (ITCC’05) - Volume II*. Vol. 1. 2005, 562–567 Vol. 1. DOI: 10.1109/ITCC.2005.213.

[41] François-Xavier Standaert and Cedric Archambeau. “Using Subspace-Based Template Attacks to Compare and Combine Power and Electromagnetic Information Leakages”. In: *Cryptographic Hardware and Embedded Systems – CHES 2008*. Ed. by Elisabeth Oswald and Pankaj Rohatgi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 411–425. ISBN: 978-3-540-85053-3.

[42] Douglas R. Stinson. *Cryptography: Theory and Practice*. 3rd ed. CRC Press, 2005.

[43] M. Strobel, M. Stieber, and M. Hutter. “The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations”. In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 1–8.

[44] Roman Vershynin. *High-dimensional probability: An introduction with applications in data science*. Vol. 47. Cambridge University Press, 2018.

[45] Lichao Wu, Guilherme Perin, and Stjepan Picek. *I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis*. Cryptology ePrint Archive, Paper 2020/1293. 2020. URL: https://eprint.iacr.org/2020/1293.

[46] Lichao Wu et al. “Leakage Model-flexible Deep Learning-based Side-channel Analysis”. In: *IACR Communications in Cryptology* 1.3 (Oct. 7, 2024). ISSN: 3006-5496. DOI: 10.62056/ay4c3txol7.

[47] Gabriel Zaid et al. “Methodology for Efficient CNN Architectures in Profiling Attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.1 (2020), pp. 1–36. DOI: 10.13154/tches.v2020.i1.1-36. URL: https://doi.org/10.13154/tches.v2020.i1.1-36.

Appendix

.1. Detailed description of SoS

.1.1. SoS layer description and hyperparameter search

It is important to firstly mention that the architecture for SoS was considered fixed in the ASCAD fixed dataset, and was subject to search in the ASCAD variable dataset. Once again, this was due to on the one hand the dataset being more complicated, but furthermore to the knowledge gained with the earlier experiments, which allowed the search space to be reduced. I will be describing the exact structure of each block, comparing the scenarios for each dataset and highlighting which parameters were searched in the ASCAD variable scenario:

Metadata embedding

The metadata embedding take as input either 2 or 3 values in the range (0,1,...,255), corresponding to plaintext, label, and key, in the case of ASCADv. These are turned into n-cryptograms by adding 256 and 512 to the second and third values, therefore creating unique words, as explained in section 4. Hence, this part of the input can be interpreted as a 3-word description about the trace information, from a vocabulary of 768 words. These are embedded into dimension 32 (ASCADf) or between 12, 24 and 36 (ASCADv). The 3 embeddings are then concatenated and normalized, constituting the output of this block, which is fed to the Language Model.

Preprocessor

In the ASCADf, the preprocessor is composed of 2 1D convolution layers with 5 filters, kernel size 40 and strides 20. Afterwards, 2 dense layers are applied.

For the ASCADv, there are 3 convolution layers with 6 filters, with kernel size 40 and strides 20, followed by either 2,3, 4 or 5 dense layers .Similarly to the ASCADf scenario, dense layers are applied. The number of layers is variable - a total number of dense layers is searched, and the DML block, the language model and the preprocessor all have half of this number of dense layers (in order for a given input to go through this number of layers in total). As suggested in [10], the number of neurons in each layer is defined by the initial dimension and final dimension, being the number of neurons in intermediate layers in geometric progression. In this case, the final dimension is searchable (output size) and the initial dimension is the size of the output of the convolution layers - which varies depending on the embedding size.

All layers are followed by normalization.

Language Model

The language model is simply dense layers followed by normalization: 2 in ASCADf and the same number as Preprocessor in ASCADv. The input is the output of metadata embedding and preprocessor, concatenated. All layers are followed by batch normalization.

Deep Metric Learner

The deep metric learner has 2 dense layers in ASCADf, and the same number of layers as preprocessor in ASCADv. The output size is 100 in ASCADf, and either 50, 100 or 200 in ASCADv. The layers are applied in parallel with the same weights to the output from the language model and the output from the preprocessoe (which are the same size) and then the 2 resulting vectors are compared with cosine similarity, resulting in a float output in [0,1].

.2. Hyperparameter search results

.2.1. PTMP ASCADf

Parameters	NOPOI			OPOI
	desync=0	desync=100	desync=200	desync=0
Convolution layers:	3	2	2	2
Filters:	(16,64,128)	(32,64)	(16,32)	(32,8)
Kernel size:	(20,25,30)	(30,20)	(30,20)	(10,15)
Stride size:	(10,5,10)	(15,15)	(5,10)	(5,5)
Activation function:	(s,s,s)	(s,s)	(s,s)	(r,s)
Pooling type:	(a,a,a)	(m,m)	(a,a)	(max,av)
Dense layers:	2	1	1	1
Neurons:	(256,50)	(256)	(50)	32
Activation function:	(s,r)	(s)	(s)	(s)
Optimizer:	adam	adam	rmsprop	adam
Learning rate:	0.0001	0.001	0.001	0.0001

Table 1: PTMP parameters for mask r_3.

Parameters	NOPOI			OPOI
	desync=0	desync=100	desync=200	desync=0
Convolution layers:	3	3	2	1
Filters:	(64,128,128)	(128,64,64)	(128,64)	8
Kernel size:	(20,30,30)	(20,15,10)	(20,10)	15
Stride size:	(15,10,15)	(5,15,10)	(15,15)	10
Activation function:	(s,r,s)	(s,s,s)	(r,r)	(r)
Pooling type:	(m,m)	(m,m,a)	(a,a)	(max)
Dense layers:	0	3	3	2
Neurons:	-	(256,100,256)	(256,50,256)	(128,64)
Activation function:	-	(r,s,s)	(s,s,r)	(s,s)
Optimizer:	adam	rmsprop	adam	rmsprop
Learning rate:	0.0001	0.0001	0.0001	0.001

Table 2: PTMP parameters for mask r_{out}.

.2.2. PTMP ASCADv

Parameters	NOPOI			OPOI	OPOI (student)
	desync=0	desync=100	desync=200	desync=0	desync=0
Convolution layers:	3	4	4	3	2
Filters:	(128,64,128)	(64,128,128,64)	(64,128,128,64)	(32,64,64)	(32,32)
Kernel size:	(20,10,20)	(30,20,10,30)	(30,20,10,30)	(10,10,20)	(30,10)
Stride size:	(5,10,15)	(5,10,15,10)	(5,10,15,10)	(10,5,10)	(15,5)
Activation function:	(r,r,r)	(r,r,s,s)	(r,r,s,s)	(s,r,r)	(r,r)
Pooling type:	(a, m, m)	(a,m,m,m)	(a,m,m,m)	(a,m,m)	(a,a)
Dense layers:	1	2	2	1	2
Neurons:	(128)	(256)	(256)	(256,128,128)	(128,256)
Activation function:	(s)	(s)	(s)	(r,r,s)	(r,r)
Optimizer:	Adam	rmsprop	rmsprop	adam	adam
Learning rate:	0.000649	0.0000283	0.0000283	0.0000198	0.000276,
Batch size:	32	32	32	32	16
Decay rate:	0.971	0.966	0.988		$\alpha = 0.66, T = 3.5$

Table 3: PTMP parameters for mask r_3 (ASCADv).

Parameters	NOPOI			OPOI	OPOI (student)
	desync=0	desync=100	desync=200	desync=0	desync=0
Convolution layers:	2	2	2	3	2
Filters:	(64, 128)	(64, 32)	(64, 64)	(64,64,32)	(32,32)
Kernel size:	(20,20)	(30,30)	(10,20)	(20,10,10)	(20,20)
Stride size:	(5,10)	(5,5)	(5,10)	(20,10,10)	(10,10)
Activation function:	(s, s)	(r,s)	(s,s)	(r,r,s)	(r,r)
Pooling type:	(m,m)	(m,m)	(m,m)	(a,a,a)	(a,a)
Dense layers:	1	2	1	2	1
Neurons:	(256)	(128,64)	(128)	(128,256)	(128)
Activation function:	(s)	(s,s)	(s)	(r,r)	(r)
Optimizer:	rmsprop	rmsprop	rmsprop	rmsprop	adam
Learning rate:	0.0000202	0.0000766	0.000173	0.00000799	0.000311
Batch size:	32	64	64	16	32
Decay rate:	0.911	0.978	0.968	0.943	$\alpha = 0.68, T = 1.64$

Table 4: PTMP parameters for mask r_{out} (ASCADv).

.2.3. SoS ASCADf

Parameters	NOPOI (desync=0)	OPOI (desync=0)
Batch size	64	64
Initial LR	0.01	0.01
Decay Rate	0.921	0.968
Loss Function	hinge	hinge

Table 5: SoS parameters attacking ASCADf using mask r_3

Parameters	NOPOI (desync=0)	OPOI (desync=0)
Batch size	64	64
Initial LR	0.01	0.01
Decay Rate	0.949	0.980
Loss Function	hinge	hinge

Table 6: SoS parameters attacking ASCADf using mask r_{out}.

Parameters	NOPOI	
	desync=100	desync=200
Batch size	128	128
Initial LR	0.01	0.01
Decay Rate	0.98	0.98
Loss Function	hinge	CE

Table 7: SoS parameters attacking ASCADf with desync using mask r_3

Parameters	NOPOI	
	desync=100	desync=200
Batch size	128	128
Initial LR	0.01	0.01
Decay Rate	0.96	0.94
Loss Function	hinge	hinge

Table 8: SoS parameters attacking ASCADf with desync using mask r_{out}

.2.4. SoS ASCADv

Parameters	NOPOI (desync=0)
Batch Size:	32
Initial LR:	0.000322
Decay Rate:	0.939
Dense Layers:	4
Embedding size:	36
Output size:	100
Loss Function:	CE

Table 9: SoS parameters attacking ASCADv using mask r_3.

Parameters	NOPOI (desync=0)
Batch Size:	32
Initial LR:	0.000130
Decay Rate:	0.988
Dense layers:	4
Embedding size:	36
Output size:	100
Loss Function:	CE

Table 10: SoS parameters attacking ASCADv using mask r_{out}.

Parameters	NOPOI (desync=0)
Batch size	32
Initial LR	0.000298
Decay rate	0.986
Dense layers	4
Embedding size	36
Output size	100
NTMP convs	3
NTMP filters	128
NTMP kernel size	10
NTMP dense layers	4
NTMP neurons	128
Loss Function	CE

Table 11: SoS parameters attacking ASCADv for NTPM configuration (changing silliness).

Parameters	NOPOI (desync=0)
Batch size	32
Initial LR	0.000194
Decay rate	0.981
Dense layers	4
Embedding size	36
Output size	200
Loss Function	CE

Table 12: SoS parameters attacking ASCADv with both masks.