Noname manuscript No.
(will be inserted by the editor)

Autoencoder-enabled Model Portability for Reducing Hyperparameter Tuning Efforts in Side-channel Analysis

Marina Krček · Guilherme Perin

Received: date / Accepted: date

Abstract Hyperparameter tuning represents one of the main challenges in deep learning-based profiling side-channel analysis. For each different side-channel dataset, the typical procedure to find a profiling model is applying hyperparameter tuning from scratch. The main reason is that side-channel measurements from various targets contain different underlying leakage distributions. Consequently, the same profiling model hyperparameters are usually not equally efficient for other targets. This paper considers autoencoders for dimensionality reduction to verify if encoded datasets from different targets enable the portability of profiling models and architectures. Successful portability reduces the hyperparameter tuning efforts as profiling model tuning is eliminated for the new dataset, and tuning autoencoders is simpler. We first search for the best autoencoder for each dataset and the best profiling model when the encoded dataset becomes the training set. Our results show no significant difference in tuning efforts using original and encoded traces, meaning that encoded data reliably represents the original data. Next, we verify how portable is the best profiling model among different datasets. Our results show that tuning autoencoders enables and improves portability while reducing the effort in hyperparameter search for profiling models. Lastly, we present a transfer learning case where dimensionality reduction might be necessary if the model is tuned for a dataset with fewer features than the new dataset. In this case, tuning of the profiling model is eliminated and training time reduced.

Keywords Side-channel Analysis, Autoencoders, Preprocessing, Hyperparameter Tuning, Portability, Transfer Learning

Marina Krček
Delft University of Technology,
Mekelweg 2, Delft, The Netherlands
E-mail: m.krcek@tudelft.nl

Guilherme Perin
Leiden University,
Rapenburg 70, Leiden, The Netherlands
E-mail: g.perin@liacs.leidenuniv.nl

1 Introduction

Hardware and software implementations of cryptographic algorithms may leak unintended and measurable side-channel information such as power consumption, electromagnetic emissions, and execution time. Although mathematically secure, these cryptographic implementations may become vulnerable to side-channel attacks (SCAs). SCA is an implementation attack mainly categorized into direct and two-stage attacks. Direct attacks, also known as non-profiled SCA, mainly consist of simple power analysis [11], differential power analysis [12], and correlation power analysis [4]. These attacks explore the statistical dependency between leaked side-channel information and secret cryptographic keys. Recovering the secret depends on running the attack over all possible key hypotheses through a divide-and-conquer strategy and selecting an efficient statistical distinguisher (e.g., Pearson correlation, difference-of-means, or mutual information). On the other hand, a two-stage, or profiling SCA [8], can evaluate the security of a cryptographic implementation by assuming a stronger adversary. Profiling SCA assumes that a potential adversary has an open device (identical to the target one) that provides conditions to learn a profiling model by reprogramming the key and input data to the cryptographic algorithm. Depending on how much knowledge is assumed that the adversary possesses (e.g., source code and access to the secret randomness of the

implementation), profiling SCA allows the deployment of worst-case (i.e., white-box) or black-box security assessment.

Countermeasures such as masking and hiding are often considered to mitigate SCA. For twenty years, Gaussian template attacks (GTA) [8] have proven to be theoretically the best option to test the worst-case security of SCA countermeasures [5]. Deep learning (DL) has been widely investigated as an alternative profiling SCA solution in the last few years. The results with real-world datasets have demonstrated that deep neural networks provide several practical advantages in comparison to GTA, such as skipping points-of-interest or feature selection from raw measurements [14, 19], relaxing assumptions about underlying leakage distribution, and being less sensitive to trace desynchronization [7, 25, 29]. However, together with large training times, the main open challenge for DL-based SCA is hyperparameter tuning. In [20], the authors suggested that hyperparameter tuning should be taken as one of the adversarial assumptions, together with the number of profiling and attack measurements. However, verifying the correctness and reliability of a DL-based profiling model concerning its hyperparameters is still difficult. Even considering advanced hyperparameter search algorithms [22, 26] cannot guarantee that the obtained best model delivers reliable security assessment.

Hyperparameter tuning is a trade-off between time effort and neural network performance, as there is no proven best way to tune the network in a reasonable time. According to [20], the maximum number of searched DL models should be considered when inferring the target's security. A profiling SCA process that is unbounded in the number of hyperparameter tuning models (or learnability capacity) would be able to deliver reliable security assessment[1]. However, as the number of searched models is always limited in reality, one would like to optimize the model search process by reducing the hyperparameter tuning efforts by ensuring that a reliable and efficient DL model is always found and trained within available computation bounds. In other words, by applying DL-based profiling SCA, the security evaluator wants to ensure that a successful security assessment (i.e., the one that fails in recovering the secret) results from an SCA-secure implementation instead of a wrong profiling attack. One way to reduce hyperparameter search effort across different targets is to apply preprocessing techniques on raw side-channel measurements, such as points-of-interest selection or dimensionality reduction.

In this paper, we consider only dimensionality reduction because points-of-interest selection tends to be inefficient due to the presence of masking countermeasures in the evaluated datasets. We assume a black-box threat model, i.e., an adversary without access to secret masks during profiling and attack phases. We consider autoencoders for dimensionality reduction, which were already considered for denoising the SCA traces and improving the profiling SCA performance [27]. Additionally, they are used in non-profiling SCA for similar preprocessing tasks [13]. Our primary goal is to verify whether efforts can be moved from tuning a profiling deep neural network model to tuning an autoencoder by reusing profiling modes across different datasets. Our main contributions are:

1. We experimentally confirm that the standard reconstruction error metric for autoencoders works well for SCA settings. Moreover, the data encoded with autoencoders stays relevant, where we show that tuning efforts on encoded data are similar to tuning on original traces.
2. We demonstrate that the portability of profiling model hyperparameters is possible. We apply the same best profiling model across different datasets encoded into the same dimension with the obtained best autoencoders. Thus, the same profiling model obtained for one dataset can be utilized for other datasets, which reduces the hyperparameter search effort.
3. We show through transfer learning that our best profiling model can be applied to different datasets, eliminating hyperparameter tuning of the profiling model and reducing training time.

The analysis provided in this paper contributes to making DL-based SCA more practical for security evaluations of cryptographic implementations when protected with the first-order Boolean masking schemes.

2 Background

2.1 Deep Learning-based Side-channel Attacks

In deep learning-based side-channel attacks (DL-based SCA), the main goal is to train deep neural network parameters θ with training data $\mathcal{D}$ by minimizing a loss function $\mathcal{L}$. Each instance of training data $\mathcal{D}$ consists of a tuple $(\mathtt{x}_i, \mathtt{y}_i)$, where $\mathtt{x}_i$ is a one-dimensional vector representing the i-th side-channel measurement (or trace) in a dataset $\mathcal{D}$. The range of i is from 0 to the

[1] Note that this conclusion only holds, at the moment, for first-order masking schemes with hiding countermeasures (e.g., desynchronization). For high-order masking schemes with or without shuffling and desynchronization, it is still an open question whether deep learning models can deliver reliable worst-case or non-worst-case security assessments (see, for instance, the discussion in [15], Section 6, for the non-worst-case assessments).

size of the dataset $|\mathcal{D}|$. The term $\mathtt{y}_i$ refers to the label (or class) associated to $\mathtt{x}_i$.

Labeling a dataset requires the definition of a leakage model and a selection function. In SCA, the main leakage models are identity (ID), Hamming weight (HW), Hamming distance (HD), and bit-level models. The identity model refers to the direct value of an intermediate being processed by a cryptographic algorithm, while HW refers to the Hamming weight of such intermediate. The HD model returns the Hamming weight from the *xor* between two intermediate variables. Bit-level models usually consider the most or least significant bit from an intermediate variable. The intermediate variable is defined according to a key-dependent selection function that usually returns an intermediate byte from the cryptographic algorithm. For the case of AES encryption, this intermediate for the i-th trace $\mathtt{x}_i$ could be an `S-Box` output byte in the first encryption round, i.e., $\mathtt{y}_i = \texttt{S-Box}(\mathtt{d}_j \oplus \mathtt{k}_j)$, where $\mathtt{d}_j$ and $\mathtt{k}_j$ are the j-th plaintext and key bytes ($j \in [0, 15]$ for a key size of 128 bits), respectively.

From the training set $\mathcal{D}$, we select a subset $\mathcal{V}$ to validate the trained model. This model is later tested on a separate dataset $\mathcal{A}$ collected from the attacked device that we refer to as the attack set. Since the goal is to obtain the secret key from $\mathcal{A}$ (or a single byte of the key), we use guessing entropy (GE) [23] to assess the attack performance. The best possible neural network model is the one that requires minimal attack complexity, which is measured in terms of the minimum number of attack traces that are necessary to successfully recover the key [6].

To compute GE, we first predict the validation or attack set and obtain class probabilities $\mathtt{p}_{i,\mathtt{y}_i}$ for each trace i. As labels $\mathtt{y}_i$ are derived from a key-dependent selection function, we obtain the log-likelihood l_k of a certain key byte $\mathtt{k}_j \in [0, 255]$:

$$l_k = \sum_{i=0}^{N_a - 1} \texttt{log}\ \mathtt{p}_{i,\mathtt{y}_i}, \tag{1}$$

where N_a is the number of traces in the predicted set. This process is then repeated for all possible key byte hypotheses. Each hypothesis will define different labels $\mathtt{y}_i$ for each trace. The key rank of the correct key $\mathtt{k}^*$ is obtained by sorting all l_k values and by returning the position of l_{k^*} associated with the correct key byte k^*. The GE of the correct key, $\mathtt{ge}^*$, is given by an empirical process in which we repeat the key rank process multiple times (each time with a different and randomly selected subset from the attack or validation set). We obtain an average log-likelihood or key guessing vector $\mathtt{g}$ and get the average position of the correct key $\mathtt{k}^*$ inside $\mathtt{g}$. When $\mathtt{ge}^* = 1$, we say that the model successfully recovers the key with N_a attack traces. The minimum number of traces to retrieve the key is referred to as $N_{\mathtt{ge}^*=1}$.

Although the primary goal of training a deep neural network in the SCA context is to minimize $N_{\mathtt{ge}^*=1}$, the models in this paper are still trained with a categorical cross-entropy loss function. In [16], the authors showed that minimizing this loss function is aligned with minimizing $N_{\mathtt{ge}^*=1}$.

2.2 Autoencoders (AEs)

Autoencoder (AE) is a specific self-supervised neural network used for data compression, dimensionality reduction, generating new data, denoising, etc. The authors in [10] first used autoencoders for dimensionality reduction. Different autoencoders, such as denoising or variational autoencoders, are described in [1, 18]. We use autoencoders for dimensionality reduction to learn, in an unsupervised manner, an informative smaller representation of the data. We consider deep autoencoders since they are often better than shallow or linear counterparts. While variational autoencoders are very popular, they are more helpful in generating new data, which is different from our goal here.

Autoencoders usually have an encoder and decoder part. The encoder takes the original input and learns a function that encodes the data into a representation given by a latent space. In dimensionality reduction, the input dimension is reduced in latent space. That middle layer is known as the "bottleneck layer", as it holds the data's compressed representation. Later, we use the decoder function to reconstruct the original input from the encoded data. Both encoder and decoder are neural networks, commonly symmetrical, having the same type and number of layers with the same layer sizes.

The objective function of the autoencoder is minimizing the difference between input and output by preserving the relevant information. The compressed data is evaluated by the decoder's ability to reconstruct the original input from the compressed data, so the common metric is Mean Squared Error (MSE). The output for autoencoders is the input itself, so MSE is calculated with

$$MSE = \frac{1}{m} \sum_{i=1}^{m} \left(\mathtt{x}_i - \hat{\mathtt{x}}_i\right)^2, \tag{2}$$

where $\mathtt{x}_i$ is the original observation and $\hat{\mathtt{x}}_i$ its reconstruction, while m is the number of inputs (samples). In SCA, $\mathtt{x}_i$ is the side-channel trace with n features, for which the distance from $\hat{\mathtt{x}}_i$ is again MSE. Therefore, we do not use labels as in profiling models and do not need

to use any leakage model. In this work, we search for the best autoencoders, following the information from [18] for defining the hyperparameter tuning space.

2.3 Transfer Learning

Transfer learning (TL) in machine learning focuses on transferring knowledge across domains and aims to leverage knowledge from a related domain to improve learning in a new task (target domain). The success of transfer learning depends on many factors, such as the relevance between the source and target domains and the learner's (model's) capacity to find transferable and valuable knowledge across the two domains. Transfer learning can be categorized based on the feature space between the two domains and the availability of the labels. More information on categorizations of TL is found in surveys, e.g., [17, 24, 30].

Our case belongs to inductive transfer learning, where we have labels for both the source and target domains (different intermediate values belonging to a specific dataset). We aim to achieve high performance in the target task. There are many approaches to transfer learning, and they depend on what we aim to transfer. In our case, we use parameter-based TL to transfer knowledge at the model/parameter level. We use models trained on one dataset and use them for different datasets. Our main objective is to obtain accurate predictions in the target domain for the new task. Specifically, we train the model to learn the correct key $\mathtt{k}^*$ of another dataset. We do it with parameter sharing so that we have a neural network for the source task, and we share (freeze) most of the layers and fine-tune the last few layers to obtain a network that works for the targeted task. We keep the first layers since the first layers in deep neural networks appear not to be specific to particular datasets or tasks [28].

2.4 Datasets

We describe three datasets that are used in our experiments.

2.4.1 DPAcontest v4.2

DPAcontest v4.2 dataset (here referred as `DPAv4.2`)[2] is the second implementation available in the DPAcontest v4 [3]. It is an improved version implemented in software on an 8-bit Atmel ATMega-163 smart card and corrects several leaks identified in its previous generation. This dataset represents the power consumption of the first AES encryption round, and the AES implementation is protected with Rotate Shift countermeasure. The dataset contains a total of 80 000 traces, and each of them contains 1 704 402 sample points. In our experiments, we trim the dataset to the interval representing the processing of the 13-th `S-box` byte, resulting in 2 000 samples per trace. The first interval ranges from sample 305 000 to 315 000 from original measurements. We apply the resampling process with a resampling window of 10 and step of 5, resulting in 2 000 samples per measurement. We use 70 000 traces for training (which contains 14 different keys).

2.4.2 ASCAD

ASCAD dataset[3] with a fixed key (`ASCADf`), along with ASCAD dataset with a random key (`ASCADr`), consists of measurements from masked AES on the 8-bit ATMega8515 MCU target without any specific hiding countermeasures activated on the target [2]. For `ASCADf` dataset, the key is fixed for all measurements. We have 50 000 training traces with 700 features per trace. `ASCADr` dataset corresponds to the second campaign with the same target and setup as in `ASCADf`. However, in this setting, the key is variable for 66% of the measurements. We use 200 000 training traces with 1 400 features per trace.

For all datasets, we use 5 000 traces for validation and another 5 000 traces as the attack set in both profiling attacks and autoencoders. We use 3 000 traces randomly chosen from that 5 000 in each key rank calculation to calculate GE.

3 Experimental Setup

In this section, we provide details about our experimental setup. The process starts with a hyperparameter search to find the best autoencoders for different datasets. Before that, we verify that the MSE metric is appropriate as it keeps the side-channel leakage in the reconstructed traces. Then we verify if searching for profiling neural network models remains similar when we train the models with the encoded datasets. We compare the attack performance of profiling models trained with encoded and original datasets. That is necessary to validate that encoded data stays relevant without worsening tuning efforts. Next, we reused profiling models' hyperparameters across multiple datasets as it was shown that tuning encoded data is equal to

[2] `https://www.dpacontest.org/v4/42_doc.php`

[3] `https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1`

tuning original datasets. We consider portability from encoded data to other encoded data and from original to encoded data. The first case enables universal models where all datasets are represented in a similar latent space. The second case is portability when we want to reuse an architecture from different feature spaces. Finally, we explore transfer learning advantages utilizing autoencoders and profiling models. To summarize, we apply the following steps:

1. Search for the best latent space size for all datasets based on two datasets.
2. Search for the best autoencoders with the lowest Mean Squared Error (MSE) by setting the best found latent space size.
3. Compare the performance of profiling models when trained with original and encoded traces of the datasets.
4. Investigate the portability of best profiling model hyperparameters trained with an *encoded dataset* to other *encoded datasets*. All datasets are encoded into the same latent dimension by using best-found autoencoders.
5. Investigate the portability of the best profiling model hyperparameters trained with an *original dataset* to other *encoded datasets*. The concept is used when a new dataset has more features than the original dataset. The new dataset is encoded into the same dimension as the original dataset using best-found autoencoders.
6. Investigate transfer learning of best profiling model trained with an *original dataset* to other *encoded datasets* encoded into the same dimension using the best autoencoders.

The overall structure of our experimental setup and the corresponding steps are shown in Figure 1. Additionally, the source code is publicly available[4].

3.1 Autoencoder Architectures

We consider the following CNN and MLP autoencoder structures:

- `ae_cnn`: autoencoders with convolution layers.
- `ae_mlp`: autoencoders given by symmetric encoder and decoder blocks, in which all layers have the same number of neurons. Latent size can be smaller, equal, or larger than the number of neurons in previous layers.
- `ae_mlp_dcr`: autoencoder with decreasing number of neurons in subsequent layers (with possible repetition). We do not ensure that the layer before the latent space is strictly larger (or equal) to the latent dimension.
- `ae_mlp_str_dcr`: autoencoder with decreasing number of neurons in subsequent layers in the encoder. Here, *_str_dcr* stands for strictly decreasing. The latent size is smaller than the number of neurons in the previous layer. However, the cases where we still use the same number of neurons in layers before the latent layer are possible.

We have several options for MLP autoencoders, while the usual, most common choice is `ae_mlp_str_dcr`. A decreasing number of neurons in the encoder and symmetrical decoder are commonly chosen because, intuitively, decreasing the number of neurons forces generalization and seems useful for dimensionality reduction. The real benefit of this structure is possibly lower computation costs compared to alternatives. However, as in classification, other options can be explored. Thus, we test the possibilities mentioned above where the number of neurons is not consistently decreasing, and the latent size is not strictly following the decreasing pattern.

In described autoencoder types, the encoder and decoder with MLP structure are always symmetrical, which means that the number of layers is the same in both encoder and decoder blocks. Also, the layer sizes are symmetrically decreasing in the encoder while increasing in the decoder. While common, this is again not strictly defined and can be explored. Intuition again says it makes the most sense for the decoder to follow a reverse structure from its encoder counterpart, but other possibilities can be similarly capable of good performance. For this setting, we keep the traditional symmetrical design.

CNN autoencoder uses similar convolutional blocks to those reported in [18]. Specifically, we use a convolutional layer followed by a pooling layer in the encoder. While they specified Max pooling, we allow both Max and Average pooling in hyperparameter selection. For the decoder, we use upsampling followed by a standard convolutional layer. Since there are more options, the convolutional autoencoder (ConvAE) structure is more complex to define than the MLP autoencoder. We observe in the literature versions of ConvAE increasing and decreasing the number of filters while kernel size and pooling size remain the same. In some cases, kernel sizes were changing. Thus, there is no specific best way to structure the ConvAE. In our case, we increase the number of filters in the encoder because the kernel size and pooling reduce the number of features, sometimes to only one. Thus, having more filters in those deeper layers ensures that after flattening, we have more than one neuron before the last fully-

[4] The code is available at `https://github.com/marinakrcek/AutoEncodersDLSCA`.

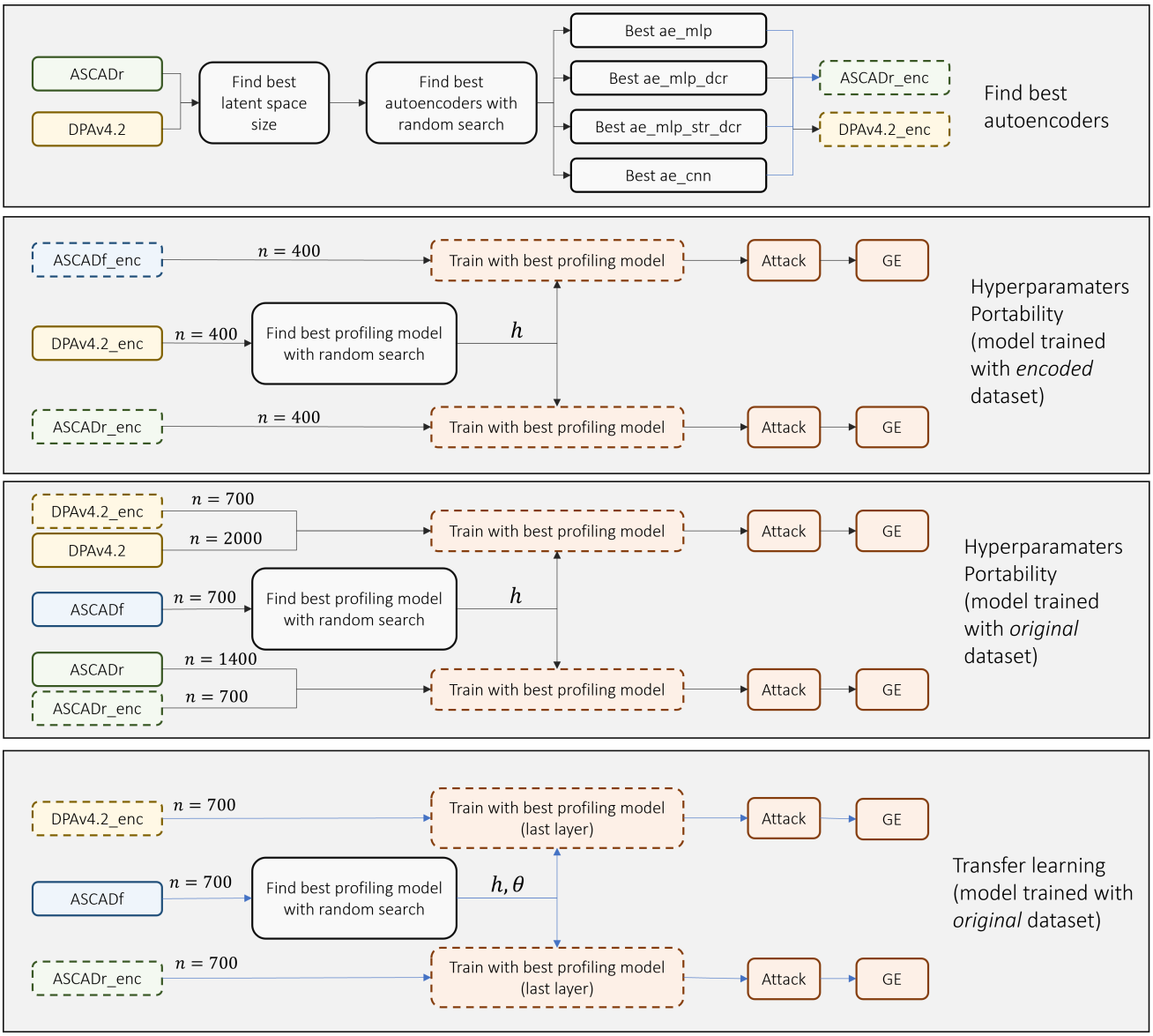


Fig. 1: Experimental setup. The term n refers to the number of features in datasets. We denote h as the set of architecture hyperparameters and θ as trainable parameters (weights and biases) in portability cases.

connected layer. We increase the number of filters per layer following the expression $nb\_filters \cdot 2^i$, where i is *the order of the layer* $+$ 1. In the decoder, with the combination of upsampling and standard convolutional layer, upsampling increases the number of features, while kernel size again decreases it. Thus, we keep the same expression for increasing the number of filters. Both the encoder and decoder end with a flattened layer followed by a fully-connected layer with the number of neurons equal to the latent size in the encoder and input size in the decoder.

In this work, we tested different structures of MLP autoencoders. At the same time, more analysis should be done for the CNN autoencoder, as the described structure is one of many possibilities. We leave this exploration on CNN structures for future work.

3.2 Autoencoder Metric Analysis

Autoencoders for dimensionality reduction imply finding a reduced representation of input data through a latent space. To assess the quality of the reduction and obtained latent representation, the most common error metric is Mean Squared Error (MSE):

$$MSE = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (\mathbf{x}_{ij} - \hat{\mathbf{x}}_{ij})^2 , \tag{3}$$

where $\mathtt{x}_{ij}$ is the j-th feature value of i-th original side-channel observation and $\hat{\mathtt{x}}_{ij}$ its reconstruction. m is the number of traces (inputs), and n is the number of features in the side-channel trace. Minimizing the MSE leads to a good reconstruction of the original input. To verify whether minimizing MSE is meaningful for SCA traces, we quantify if the leakage is still preserved in the reconstructed traces by calculating the Signal-to-Noise Ratio (SNR). SNR is computed as a leakage assessment of side-channel measurements according to a preselected intermediate variable. Since evaluated datasets in this paper were collected from first-order masking AES implementations, first-order intermediate values (such as $\mathtt{S\text{-}Box}(\mathtt{d}_j \oplus \mathtt{k}_j)$) show no significant leakages. Thus, we compute SNR to verify the occurrence of leakages for the masked $\mathtt{S\text{-}Box}$ output intermediate values[5], i.e., $v = \mathtt{S\text{-}Box}(\mathtt{d}_j \oplus \mathtt{k}_j) \oplus \mathtt{m}_i$, where $\mathtt{m}_i$ is the mask of the i-th trace. For that, we compute the mean and variance side-channel traces for a group of traces represented by a specific intermediate variable $v \in [0, 255]$:

$$\mu_v = \frac{1}{N_v} \sum_{i=0}^{n_v-1} \mathtt{x}_i^v \tag{4}$$

$$\sigma_v^2 = \frac{1}{N_v - 1} \sum_{i=0}^{N_v-1} \left(\mathtt{x}_i^v - \mu_v\right)^2, \tag{5}$$

where N_v is the number of side-channel traces represented or labeled with intermediate variable v. Next, we obtain the mean vector from all 256 variance vectors σ_v^2:

$$\mu_\sigma = \frac{1}{256} \sum_{v=0}^{255} \sigma_v^2 \tag{6}$$

and the variance of mean vectors μ_v:

$$\sigma_\mu = \frac{1}{255} \sum_{v=0}^{255} \left(\mu_v - \mu_\sigma\right)^2. \tag{7}$$

Finally, SNR is given by:

$$SNR = \frac{\sigma_\mu}{\mu_\sigma}. \tag{8}$$

The SNR from Eq. (8) results in a vector with the same length as side-channel traces. We compute this vector for original and reconstructed traces. Then, we take the maximum SNR peak obtained with original traces and subtract it from the value on that exact location in the SNR obtained from reconstructed traces. In the result figures, we refer to this as `SNR diff`.

[5] Although our profiling attacks in later sections are all executed in a black-box manner, here we assume the knowledge of the masks only to assess if MSE metric is consistent.

4 Experimental Results

4.1 Autoencoders Search

In this section, we deploy a random search to find the best latent space size for autoencoders based on experiments with two datasets. After defining the best latent space size, we deploy a random search to find the best autoencoder architecture for MLP and CNN-based structures. We also obtain the best autoencoder types. Datasets are then encoded with these best autoencoders. Finally, we deploy another random search to find the best profiling model trained with the encoded datasets. We include the third dataset later for portability experiments while also evaluating how well the decisions made on two other datasets for latent size and autoencoder type apply to new datasets.

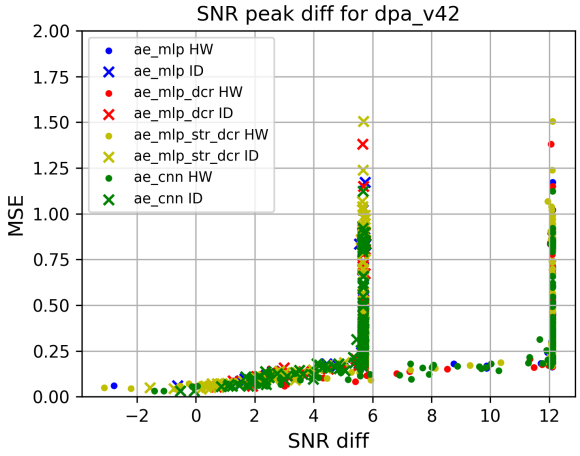
4.1.1 Assessing MSE Metric with SNR

We conduct a random search on autoencoders using MSE as the loss function for achieving good reconstruction from the latent space. In these experiments, we consider SNR to verify that minimizing MSE is a meaningful objective when tuning autoencoder hyperparameters. Here, we are not searching for the best latent space size, so we fix the latent dimension to 100 features in all cases to evaluate the MSE metric.

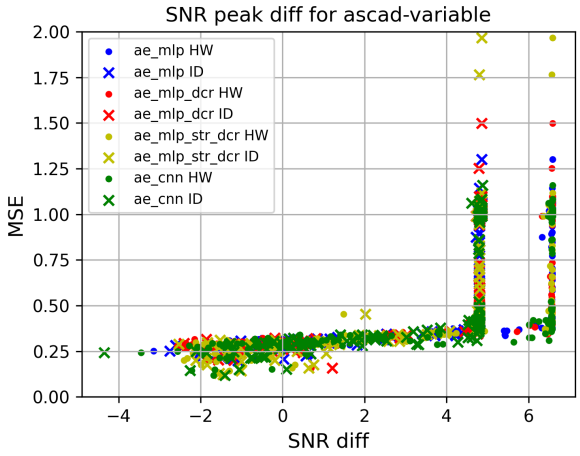
Hyperparameter search space for all autoencoder types are listed in Tables 19 and 20 in Appendix A. We randomly search for 20 models with each of the four autoencoder types and calculate the SNR difference, as described before, between SNR vectors obtained from original and reconstructed traces. The analysis is conducted for the Hamming weight and identity leakage models with $v = \mathtt{S\text{-}Box}(\mathtt{d}_j \oplus \mathtt{k}_j) \oplus \mathtt{m}_i$ as the intermediate variable to compute SNR.

Results are shown in Figure 2 for the `DPAv4.2` and `ASCADr` datasets. The x-axis in Figures 2a and 2b shows the maximum peak SNR value difference between the original and reconstructed traces. The corresponding MSE value for each autoencoder is on the y-axis. These figures show that MSE increases as the SNR peak difference increases regardless of the autoencoder type and leakage models in SNR calculations. The vertical lines occur when the reconstructed traces result in insignificant SNR peak values, indicating that side-channel leakages concerning $v = \mathtt{S\text{-}Box}(\mathtt{d}_j \oplus \mathtt{k}_j) \oplus \mathtt{m}_i$ are not preserved in the reconstructed traces. Negative SNR difference values on the x-axis indicate that the reconstructed trace has a higher SNR value than the original trace on the same sample point, which means that the corresponding autoencoder is preserving and even

amplifying the occurrence of side-channel leakages concerning v. However, MSE is not created to lead the AE to amplify any such SNR peak, as it gets minimized by correct reconstruction of the trace without amplifications.



(a) DPAv4.2 dataset.



(b) ASCADr dataset.

Fig. 2: Relation between MSE and SNR difference.

Next, we consider Pearson correlation coefficient ρ to test whether there is a positive (linear) correlation between MSE and SNR differences. Indeed, from the results in Table 1, we see a high correlation until the vertical lines (maximum difference in the SNR values). Specifically, for both datasets, the correlation is stronger for MSE below 0.5. For the DPAv4.2 dataset, the maximum correlation is for MSE values below 0.25, and in the case of ASCADr, below 0.5. Therefore, we conclude that minimizing MSE is a meaningful objective error function to optimize autoencoder models for the given datasets.

Table 1: Pearson correlation coefficient ρ and p-value for testing non-correlation. LM stands for leakage model.

Dataset	MSE	LM	ρ	p-value
DPAv4.2	> 0	HW	0.12	2.32e-03
		ID	0.12	2.67e-03
	< 0.5	HW	0.76	8.85e-68
		ID	0.76	1.62e-65
	< 0.375	HW	0.83	1.03e-80
		ID	0.83	4.35e-78
	< 0.25	HW	**0.93**	**2.04e-78**
		ID	**0.93**	**1.32e-80**
ASCADr	> 0	HW	0.03	4.84e-01
		ID	0.03	5.01e-01
	< 0.5	HW	**0.87**	**3.92e-112**
		ID	**0.86**	**6.58e-106**
	< 0.375	HW	0.68	1.05e-34
		ID	0.70	3.10e-37
	< 0.25	HW	-0.10	5.16e-01
		ID	0.01	9.37e-01

4.1.2 Searching for the Best Latent Space Size

In this section, we use random search to compare different latent space sizes. The latent sizes we consider are 20, 40, 50, 100, 200, 250, 400, and 500 for all autoencoder types except that for the ae_mlp_str_dcr, we do not use the latent size 500 as we also limit the search to 400 neurons per layer (see Table 19). By choosing these latent sizes, we ensure that the bottleneck layer in the autoencoder is always smaller than the input layer (which contains the same number of units as the input side-channel trace dimension). The datasets evaluated in this section contain 2 000 features (DPAv4.2) and 1 400 features (ASCADr), which is significantly larger than chosen latent space sizes given by the bottleneck layer. Hyperparameter search space (Tables 19 and 20) is the same as in the metric analysis provided in the previous section. The hyperparameters for the autoencoders are chosen at random, and we train 20 autoencoder models per latent size, dataset, and autoencoder type combination. The total number of autoencoder combinations in this search is 62.

The main idea here is to verify if a specific latent space size tends to provide the lowest MSE among the searched ones regardless of the dataset and autoencoder (AE) type. Considering our two datasets and four autoencoder types, we have eight cases, each testing eight or seven latent sizes. Autoencoder type ae_mlp_str_dcr does not use latent size 500, which leads to trying seven latent sizes instead of eight. We apply the following procedure to obtain the best latent size:

1. For each of these 62 combinations, we extract the autoencoder (out of 20) with the lowest MSE for that dataset, autoencoder type, and latent size.
2. For each autoencoder type and dataset combination, we rank the latent sizes based on the best (lowest) MSE. The latent size for the model with the lowest MSE gets the rank 1 being the best one.
3. For each of the latent space sizes, we average these eight ranks coming from the dataset-autoencoder (AE) type combination.

Table 2 shows the average ranks of the latent sizes. The left side of the table is for AE types except for `ae_mlp_str_dcr` as that one must have a decreasing structure, and with the latent size of 500, we cannot achieve that as we limited our number of neurons to a maximum of 400. On the right side of Table 2, we order all latent sizes according to the average rank, except for 500, and include the results from `ae_mlp_str_dcr`.

Following, we use the Friedman test [9] across all autoencoder types and the two datasets (`ASCADr` and `DPAv4.2`). This test determines whether there is a statistically significant difference between the means of three or more groups in which the same subjects appear in each group. In our case, groups are based on latent sizes, and subjects are dataset-AE type combinations. The comparison is based on the lowest MSE obtained. Friedman test calculates test statistic $Q = \frac{12}{nk(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1)$, where n is the number of subjects, k is the number of groups, and R_j is the sum of the ranks for sample j. We determine the critical value from the Chi-Square distribution table with $k-1$ degrees of freedom. Q value has to be greater than the critical value of Q for a selected significance level α to reject the null hypothesis. Commonly, significance level α of 0.05 works well [21]. The p-value is the probability of obtaining test results at least as extreme as a result observed under the assumption that the null hypothesis is correct. The null hypothesis for the Friedman test is that the mean of the groups is the same. A very small p-value means such an extreme observed outcome would be very unlikely under the null hypothesis. The null hypothesis can be rejected if the p-value is below α.

We report the Friedman test results at the bottom row in Table 2. Since the p-value is below 0.05, we conclude that the difference between the mean values of the groups (latent sizes) is statistically significant. Additionally, the test statistic Q on the left part is greater than the critical value of 14.07 for the degree of freedom 7. On the right side, the test statistic is greater than the critical value 12.59 for $\alpha = 0.05$ and degree of freedom 6. We perform the Nemenyi post-hoc test to determine which groups have different means. The results are shown in Appendix B in Tables 23 and 24 corresponding to the cases without and with `ae_mlp_str_dcr` model. The values in the tables are p-values where if the value is below 0.05, the two groups (column-row combination) have statistically significantly different means. The lowest MSE values for models with latent sizes 400 and 200 differ significantly from models with lower latent sizes (20, 40, and 50). For latent sizes 100 and 250, the difference is less significant. However, we still select only latent sizes 200 and 400 to find the best autoencoders using a random search.

Table 2: Average ranks for each latent space size.

Latent space size	w/o `ae_mlp_str_dcr`	Latent space size	with `ae_mlp_str_dcr`
400	1.5	400	1.375
200	1.83	200	2
500	3.67	250	3.375
100	4	100	3.5
250	4.3	50	5.25
50	6.33	40	5.75
40	6.67	20	6.75
20	7.67		
Q: 35.17, p-value: 1.04e-5		Q: 40.66, p-value: 3.38e-7	

4.1.3 Selecting the Best Autoencoders

After we found the best latent size for the `ASCADr` and `DPAv4.2` datasets, we randomly search for additional 80 autoencoder models to obtain a total of 100 models for latent space sizes of 200 and 400. The hyperparameter search space for each autoencoder type stays the same as in the search for the best latent size. From these 100 autoencoder models, we select the best autoencoder for each dataset. Table 3 shows the MSE of the best autoencoder for each of the given latent sizes (200 or 400) and autoencoder types per dataset. We see that for `ASCADr`, latent size 400 always results in a lower MSE. For `DPAv4.2`, `ae_mlp` and `ae_mlp_dcr` had better results with 200 features in latent space. However, we can conclude that the best autoencoder types are `ae_cnn` and `ae_mlp_str_dcr`, with the lowest MSE in both datasets using latent size 400.

Since we initially only allow up to 400 neurons per layer, with a latent space size of 400, the autoencoder `ae_mlp_str_dcr` type could not create a bottleneck architecture with decreasing number of neurons in consecutive layers of the encoder. Thus, we repeated the

Table 3: Best autoencoders. The rows are sorted based on the MSE, so the latent size order in rows is not fixed.

Dataset	AE type	Latent size	MSE
DPAv4.2	ae_mlp	200	0.053842735
		400	0.068908036
	ae_mlp_dcr	200	0.053061113
		400	0.071744457
	ae_mlp_str_dcr	400	0.033211511
		200	0.042860519
	ae_cnn	400	**0.026164241**
		200	0.057221718
ASCADr	ae_mlp	400	0.177198691
		200	0.198677342
	ae_mlp_dcr	400	0.133906147
		200	0.194515368
	ae_mlp_str_dcr	400	0.121792312
		200	0.196267218
	ae_cnn	400	**0.118710345**
		200	0.209023259

random search for another 100 models for this autoencoder type by allowing layers with 500 and 600 neurons. Table 4 shows the minimum, mean, median, and maximum MSE found in 100 models for the two datasets. Note that this table shows MSE results when the autoencoder contains layers with 400 neurons and MSE results when layers can include 400, 500, and 600 neurons.

Table 4: Autoencoder ae_mlp_str_dcr with latent space size of 400. Values are calculated on MSE from a random search of 100 different models, and the number of neurons represents the allowed values from the hyperparameter search space.

Dataset	Nb. neurons	min	mean	median	max
DPAv4.2	400	**0.03321**	**0.5295**	**0.3889**	**3.68**
	400, 500, 600	0.03373	0.8548	0.4529	27.67
ASCADr	400	0.12179	**0.6028**	**0.4701**	**2.01**
	400, 500, 600	**0.12107**	0.7988	0.4832	7.71

For DPAv4.2 dataset, an autoencoder with up to 400 neurons per layer results in a lower MSE than when we allow 400, 500, and 600 neurons per layer. The hyperparameters of the best models for both cases are in Table 5. Note that this architecture has one hidden layer with 400 neurons between the input layer and the layer with the specified latent size, and the decoder is symmetrical. When allowing 500 and 600 neurons in the random search, the best-found autoencoder has an architecture with two hidden layers with 400 neurons in the encoder and decoder. They also differ in batch size, activation function, learning rate, and weight initialization, but the optimizer is the same. The best model in the second case is not using a larger number of neurons in the first layers.

The autoencoder for the ASCADr dataset has a lower minimal MSE when the number of neurons includes 500 and 600 neurons in the random search. However, the ability to use a larger number of neurons in layers closer to the input layer was not utilized. In both cases for ASCADr dataset, the best model has the same architecture: one hidden layer with 400 neurons for the encoder and decoder and a bottleneck layer with 400 neurons. They differ in activation function and weight initialization, while batch size, learning rate, and optimizer are the same. As the best autoencoders have the same architecture (layers and neurons) for both datasets, the slight difference in the performance comes from the other hyperparameters.

In general, the results in Table 4 indicate that a random search only including the option of 400 neurons per layer delivers better MSE values than when we allow more neurons per layer. Mean, median, and maximum MSE are always lower when only 400 neurons are permitted. These results are confirmed for both datasets.

Table 5: MLP autoencoders for DPAv4.2 and ASCADr with latent space size of 400 with different allowed numbers of neurons in layers.

Nb. neurons	Hyperparameters	MSE
	DPAv4.2	
400	ae_mlp_dpav42_best: architecture: [400], batch_size: 200, activation: tanh, learning_rate: 0.0001, weight_init: he_normal, optimizer: Adam	0.03321
400, 500, 600	architecture: [400, 400], batch_size: 100, activation: elu, learning_rate: 0.001, weight_init: glorot_normal, optimizer: Adam	0.03373
	ASCADr	
400	ae_mlp_ascadr_best: architecture: [400], batch_size: 100, activation: elu, learning_rate: 0.0001, weight_init: random_uniform, optimizer: RMSprop	0.12179
400, 500, 600	architecture: [400], batch_size: 100, activation: selu, learning_rate: 0.0001, weight_init: random_normal, optimizer: RMSprop	0.12107

Based on these results, the best autoencoder we use in further experiments is the MLP autoencoder for the DPAv4.2 dataset with an MSE of 0.03321. For the ASCADr dataset, we use the MLP autoencoder with an MSE of 0.12179. The autoencoder with the CNN structure achieves even better MSE - with 0.026 for DPAv4.2, and 0.1187 for ASCADr. The hyperparameters for the best ae_cnn autoencoders are in Table 6. We use these four autoencoders in the further experiments, which are denoted ae_mlp_dpav42_best, ae_mlp_ascadr_best, ae_cnn_dpav42_best, and ae_cnn_ascadr_best.

Table 6: Best ae_cnn autoencoders for DPAv4.2 and ASCADr with latent space size of 400.

Dataset	Hyperparameters	MSE
DPAv4.2	ae_cnn_dpav42_best: batch_size: 200, filters: 16, kernel_size: 10, strides: 5, pool_size: 2, pool_strides: 2, pooling_type: Avg, conv_layers: 1, activation: tanh, learning_rate: 0.0001, weight_init: random_normal, optimizer: Adam	0.02616
ASCADr	ae_cnn_ascadr_best: batch_size: 200, filters: 16, kernel_size: 20, strides: 5, pool_size: 2, pool_strides: 2, pooling_type: Avg, conv_layers: 1, activation: tanh, learning_rate: 0.001, weight_init: random_uniform, optimizer: Adam	0.11871

4.2 Are Encoded Datasets as Good as Original Datasets?

After defining the best ae_mlp_str_dcr and ae_cnn autoencoder structures for DPAv4.2 and ASCADr datasets, we investigate if the encoded datasets can keep relevant leakage information when they are considered as training and attack datasets.

We run a random search to find different MLP and CNN profiling models, and we compare the search performance using original and encoded traces obtained from the best autoencoders listed in Tables 5 and 6. The hyperparameter search space for the profiling models is shown in Table 21. Training, validation, and attack sets are labeled with S-box$(d_2 \oplus k_2)$ (third S-box output byte in first AES encryption round[6]) for ASCADr and S-box$(d_{12} \oplus k_{12})$ (13-th S-box output byte in the first AES encryption round) for DPAv4.2. We consider the Hamming weight (HW) and identity (ID) leakage models. We search for 100 models for each combination of the leakage model and profiling model type (MLP-ID, MLP-HW, CNN-ID, and CNN-HW). We measure how many out of the random 100 models reach $\mathrm{ge}^* = 1$ for a given number of validation traces. With that information, we compare if we can more easily obtain a good model using original or encoded traces within the same hyperparameter search space. If we can get a similar amount of models out of 100 that reach $\mathrm{ge}^* = 1$ with original and encoded traces, it means that encoded traces preserve enough information and can be used for training profiling models in SCA. Results for both datasets are shown in Figure 3.

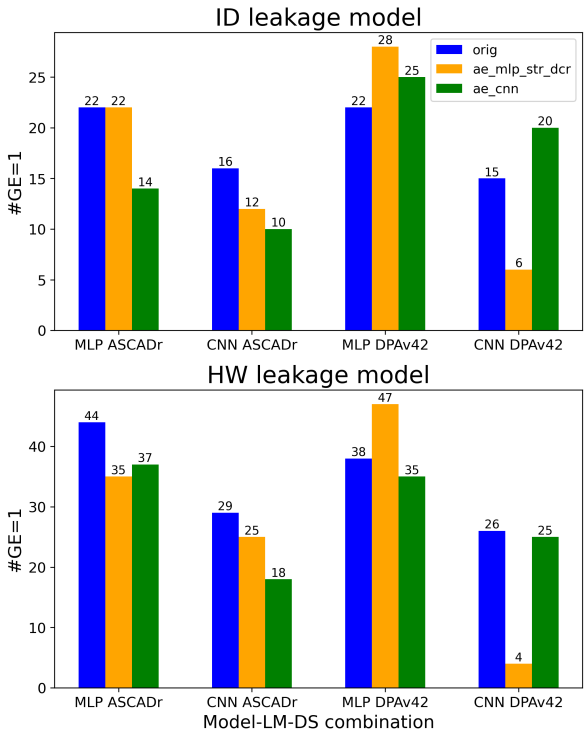


Fig. 3: Results on tuning effort using original and encoded traces. We compare the number of models reaching $\mathrm{ge}^* = 1$ out of 100 trained models with different hyperparameters selected using random search.

From the results, we see that out of 100 models, for the ASCADr dataset, only in the case with MLP and the ID leakage model we obtained the same number of models with $\mathrm{ge}^* = 1$. Looking at the number of traces $N_{\mathrm{ge}^*=1}$, using original traces on average 1 462.8 traces are necessary, while for the dataset encoded with ae_mlp_ascadr_best, we need on average $N_{\mathrm{ge}^*=1} = 1205.9$ traces. For other attack setups, using original traces led to more models with $\mathrm{ge}^* = 1$. However, using encoded data was not much worse.

[6] Note that we start counting from byte index 0.

On the other hand, for DPAv4.2, in three out of four attack settings, we obtained more models with $ge^* = 1$ when using traces encoded with ae_mlp_dpav42_best or ae_cnn_dpav42_best. Those cases are the MLP profiling model with both leakage models and the CNN profiling model with the ID leakage model. The result with the CNN profiling model and HW leakage model is again close in performance for encoded data with ae_cnn_dpav42_best and original traces.

We use the Friedman test on the eight scenarios presented in Figure 3. We want to see if there is a statistical difference between training on encoded and original traces. We obtain a test statistic of 2.7742 and a p-value of 0.2498. Thus, there is no significantly better setup based on the number of models reaching $ge^* = 1$ out of 100 runs. The initial hypothesis for Friedman is that there is no statistically significant difference in the mean of these numbers. Since the p-value, in this case, is not below 0.05, all the setups lead to similar performance. To conclude, using original traces is not statistically significantly better than using encoded data, meaning that encoded data preserves relevant features that can be used in a profiling attack.

4.3 The Portability of Profiling Models

In this section, we verify the efficiency of the best profiling model (obtained in the previous section) when found through hyperparameter tuning with one dataset but concerning different datasets. We include a third dataset to show portability from one to two other datasets. This way, we can answer the following question: *can we move effort from profiling model tuning into autoencoder tuning to reuse the same profiling model across multiple encoded side-channel datasets?*

4.3.1 Portability of Encoded-Data Trained Profiling Model to Different Encoded Datasets

We start by verifying the portability of a best-found profiling model trained with an encoded dataset concerning other encoded datasets. That is possible because we encode all datasets into the same encoding dimension, i.e., all encoded datasets contain an equal number of features. We take the best MLP and CNN profiling models trained with encoded DPAv4.2 dataset for both leakage models. Their hyperparameters and attack performance are shown in Tables 7 and 8. We test the performance of those architectures on the encoded ASCADr and ASCADf datasets.

For ASCADr, we already have the best autoencoders with the latent size of 400 (see Tables 5 and 6 for hyperparameters and MSE). We additionally train autoencoders ae_mlp_str_dcr and ae_cnn for the ASCADf dataset to encode it to 400 features per trace as well. The hyperparameters range to find the best ae_cnn autoencoder with the random search are the same as considered for the DPAv4.2 and ASCADr datasets (see Table 20). To find the best ae_mlp_str_dcr autoen-

Table 7: Best MLP and CNN profiling models obtained for the encoded DPAv4.2 dataset when encoded with the best-found ae_mlp_dpav42_best.

Model	LM	Hyperparameters	ge^*	$N_{ge^*=1}$
MLP	ID	layers: 2, neurons: 100, batch_size: 200, activation: selu, learning_rate: 0.005, weight_init: random_uniform, optimizer: Adam	1	3
	HW	layers: 3, neurons: 40, batch_size: 200, activation: selu, learning_rate: 0.0025, weight_init: glorot_uniform, optimizer: RMSprop	1	29
CNN	ID	neurons: 50, batch_size: 100, layers: 1, filters: 16, kernel_size: 20, strides: 5, conv_layers: 4, activation: selu, learning_rate: 0.005, weight_init: random_normal, optimizer: Adam	1	65
	HW	neurons: 20, batch_size: 100, layers: 2, filters: 12, kernel_size: 40, strides: 15, conv_layers: 5, activation: elu, learning_rate: 0.001, weight_init: glorot_uniform, optimizer: RMSprop	1	819

Table 8: Best MLP and CNN profiling models obtained for the encoded DPAv4.2 dataset when encoded with the best-found ae_cnn_dpav42_best.

Model	LM	Hyperparameters	ge^*	$N_{ge^*=1}$
MLP	ID	layers: 6, neurons: 200, batch_size: 200, activation: selu, learning_rate: 0.0005, weight_init: random_uniform, optimizer: RMSprop	1	2
	HW	layers: 6, neurons: 50, batch_size: 200, activation: elu, learning_rate: 0.0025, weight_init: glorot_normal, optimizer: RMSprop	1	16
CNN	ID	neurons: 200, batch_size: 400, layers: 1, filters: 12, kernel_size: 30, strides: 5, conv_layers: 4, activation: selu, learning_rate: 0.0025, weight_init: glorot_normal, optimizer: RMSprop	1	3
	HW	neurons: 200, batch_size: 400, layers: 2, filters: 12, kernel_size: 40, strides: 15, conv_layers: 5, activation: elu, learning_rate: 0.0005, weight_init: random_normal, optimizer: RMSprop	1	18

coder for the ASCADf dataset, we again consider the random search settings shown in Table 19. However, we allow the number of neurons per layer for a latent size of 400 to be [400, 500, 600, 700]. The hyperparameters for best autoencoders ae_mlp_ascadf_best and ae_cnn_ascadf_best for ASCADf with latent size 400 are reported in Table 9.

Table 9: Best autoencoders for ASCADf with latent space size of 400.

AE type	Hyperparameters	MSE
	ASCADf	
ae_mlp_str_dcr	ae_mlp_ascadf_best: architecture: [400, 700], batch_size: 200, activation: tanh, learning_rate: 0.0001, weight_init: random_normal, optimizer: Adam	0.01245
ae_cnn	ae_cnn_ascadf_best batch_size: 200, filters: 16, kernel_size: 20, strides: 10, pool_size: 4, pool_strides: 2, pooling_type: Avg, conv_layers: 1, activation: tanh, learning_rate: 0.0001, weight_init: he_uniform, optimizer: Adam	0.01380

Table 10 shows the attack performance of the best CNN and MLP profiling architectures on DPAv4.2 from Tables 7 and 8 when trained with the encoded ASCADr and ASCADf datasets. The results in this table indicate the number of times (out of 100) that the profiling model reaches $ge^* = 1$ for each scenario (leakage model, profiling model type, and autoencoder type). Before the training, we performed standardization on the encoded datasets. Standardization is a typical preprocessing method before training in the SCA and other domains. However, later we also test without standardization to observe the effects.

The results with encoded ASCADr indicate superior performance compared to results obtained with the encoded ASCADf. Performance with the encoded ASCADr for MLP with the identity leakage model when this architecture was found with ae_cnn_dpav42_best-encoded DPAv4.2 dataset is slightly worse with finding 19 and 51 models out of 100 reaching $ge^* = 1$. As mentioned, the results with encoded ASCADf are not good, specifically for cases with ASCADf encoded with ae_mlp_ascadf_best. The poor performance might come from the fact that the features in encoded data do not share comparable features despite the equal latent size. We observe that the architecture of that autoencoder is different from the architectures for the other two datasets. To improve these results for encoded ASCADf, and since for ASCADf we allowed more than 400 neurons per layer (which was not the case for other datasets), we again train 100 ae_mlp_str_dcr autoencoders for ASCADf but with only 400 neurons per layer and latent size 400. This way, the autoencoder architecture will be more similar to autoencoders of other datasets. The resulting features also become more comparable, which could improve the performance. The hyperparameters of the best autoencoder for this case are in Table 11.

Table 10: Portability results with best MLP and CNN models obtained with the encoded DPAv4.2 datasets (from ae_mlp_dpav42_best and ae_cnn_dpav42_best). Datasets ASCADr and ASCADf are encoded with their respective best autoencoders. In these results, before training, we use standardization. The training is done 100 times, and the reported number is the number of times we reach $ge^* = 1$.

Model	LM	encoded DPAv4.2 with ae_mlp_dpav42_best		encoded DPAv4.2 with ae_cnn_dpav42_best	
		ae_cnn_*_best	ae_mlp_*_best	ae_cnn_*_best	ae_mlp_*_best
		encoded ASCADr			
MLP	ID	100	100	19	51
	HW	97	97	89	73
CNN	ID	79	65	99	100
	HW	98	100	100	99
		encoded ASCADf			
MLP	ID	35	24	17	0
	HW	0	0	10	5
CNN	ID	97	5	30	0
	HW	16	0	66	1

Table 11: Best ae_mlp_str_dcr autoencoder for ASCADf with latent space size of 400, allowing only 400 neurons per layer.

Hyperparameters	MSE
ae_mlp_ascadf_best: architecture: [400], batch_size: 100, activation: selu, learning_rate: 0.0001, weight_init: he_normal, optimizer: RMSprop	0.01345

The results using the ae_mlp_ascadf_best-encoded ASCADf from the described search are in Table 12. Here, we see an improvement, which indicates our hypothesis on the similarity of latent representations with DPAv4.2

and ASCADr could be true. Accordingly, this is a crucial remark to consider if universal models are to be considered. As the feature space is more similar, the portability becomes easier. Despite the MSE being slightly worse than before, the attack performance is better since the representations are more comparable.

Table 12: Results with using encoded ASCADf from ae_mlp_ascadf_best with only 400 neurons per layer. We use standardization of the encoded dataset when training the profiling model 100 times. The number represents the number of times we reach $ge^* = 1$.

Model	LM	Best model for DPAv4.2 encoded with	
		ae_mlp_dpav42_best	ae_cnn_dpav42_best
MLP	ID	69	37
	HW	0	2
CNN	ID	100	100
	HW	51	17

After improving ae_mlp_ascadf_best, we also test best MLP and CNN profiling architectures from Tables 7 and 8 without data standardization. The results are in Table 13 and show that for ASCADr, we have similar successful behavior in comparison to results from Table 10 when data standardization was done. For encoded ASCADf, we compare results with Table 12 for ae_mlp_ascadf_best as that autoencoder was used for encoding as it was shown to be better. Additionally, we compare it with Table 10 for ae_cnn_ascadf_best. Results with and without standardization for ASCADf are also similar.

Our analysis demonstrates that reusing profiling models trained on an encoded dataset is possible. That reduces hyperparameter tuning efforts when considering new encoded datasets, where the effort is moved to tuning the autoencoder. Additionally, universal profiling architecture is then something we can consider on autoencoder-encoded data. Moreover, tuning autoencoders is easier as optimization of MSE is more straightforward.

4.3.2 Portability of Original-Data Trained Profiling Model to Different Original and Encoded Datasets

In this section, we test the portability of a best-found profiling model architecture (from random search) when it is trained on an original (i.e., not encoded) dataset. For that, we consider ASCADf, which contains 700 features. We made this choice because ASCADf has fewer features than ASCADr and DPAv4.2, which contain 1 400 and 2 000 features, respectively, in their original versions. In this case, to reuse that architecture, we need to decrease the number of features of other datasets to the size of the data used in training. However, since the input layer is a dedicated first layer in neural networks, to reuse the architecture, we can also replace that first layer. In that case, we can keep the original number of features of the new datasets. Our goal is to verify if the best-found profiling architecture with ASCADf also provides good attack performance when trained with the encoded and original ASCADr and DPAv4.2 datasets. Therefore, we have three cases per dataset - using original and encoded data with two different AE types.

Table 13: Portability results with best MLP and CNN models obtained with encoded DPAv4.2 datasets (from ae_mlp_dpav42_best and ae_cnn_dpav42_best). Datasets ASCADr and ASCADf are encoded with their respective best autoencoders. We use encoded data directly, without standardization. The training is done 100 times, and the reported number is the number of times we reach $ge^* = 1$.

Model	LM	encoded DPAv4.2 with ae_mlp_dpav42_best		encoded DPAv4.2 with ae_cnn_dpav42_best	
		ae_cnn_*_best	ae_mlp_*_best	ae_cnn_*_best	ae_mlp_*_best
encoded ASCADr					
MLP	ID	75	100	0	73
	HW	98	93	100	87
CNN	ID	87	83	100	100
	HW	97	100	100	99
encoded ASCADf					
MLP	ID	0	83	100	11
	HW	12	3	0	2
CNN	ID	98	100	43	100
	HW	29	85	1	86

Since this time we have to encode ASCADr and DPAv4.2 into 700 features, we again run a hyperparameter search to find the best autoencoders, which are reported in Table 14 with their corresponding MSE values. The hyperparameter search spaces are shown in Tables 20 and 22. Table 15 shows the best MLP and CNN profiling architectures found for the original ASCADf dataset. We ran a random search for 100 models using hyperparameter search space from Table 21.

Table 14: Autoencoders for DPAv4.2 and ASCADr with latent space size of 700.

AE type	Hyperparameters	MSE
	DPAv4.2	
ae_mlp_str_dcr	ae_mlp_dpav42_best_700: architecture: [1400], batch_size: 100, activation: tanh, learning_rate: 0.0001, weight_init: he_normal, optimizer: Adam	0.017
ae_cnn	ae_cnn_dpav42_best_700: batch_size: 200, filters: 16, kernel_size: 10, strides: 5, pool_size: 4, pool_strides: 2, pooling_type: Avg, conv_layers: 1, activation: elu, learning_rate: 0.001, weight_init: random_uniform, optimizer: Adam	0.013
	ASCADr	
ae_mlp_str_dcr	ae_mlp_ascadr_best_700: architecture: [900], batch_size: 200, activation: selu, learning_rate: 1e-05, weight_init: random_uniform, optimizer: RMSprop	0.052
ae_cnn	ae_cnn_ascadr_best_700: batch_size: 400, filters: 8, kernel_size: 20, strides: 5, pool_size: 2, pool_strides: 4, pooling_type: Avg, conv_layers: 1, activation: elu, learning_rate: 0.001, weight_init: he_uniform, optimizer: RMSprop	0.141

Table 15: Best profiling models for ASCADf.

Model	LM	Hyperparameters	ge^*	$N_{ge^*=1}$
MLP	ID	layers: 4, neurons: 40, batch_size: 400, activation: relu, learning_rate: 0.001, weight_init: random_uniform, optimizer: Adam	1	151
	HW	layers: 4, neurons: 20, batch_size: 100, activation: elu, learning_rate: 0.001, weight_init: random_normal, optimizer: RMSprop	1	1476
CNN	ID	neurons: 50, batch_size: 200, layers: 2, filters: 4, kernel_size: 10, strides: 5, conv_layers: 1, activation: relu, learning_rate: 0.0001, weight_init: random_uniform, optimizer: RMSprop	1	265
	HW	neurons: 200, batch_size: 400, layers: 1, filters: 4, kernel_size: 40, strides: 5, conv_layers: 4, activation: selu, learning_rate: 0.005, weight_init: glorot_uniform, optimizer: RMSprop	1	1734

Since we reuse only the architecture and not the trained parameters (weights and biases), we modify the input layer to use the original ASCADr and DPAv4.2 datasets that have more features than the original ASCADf. This way, we take the best architectures from Table 15 and train them with the original ASCADr and DPAv4.2 datasets as well as with their encoded versions by using the best-found autoencoders listed in Table 14. For each dataset, profiling model architecture, and leakage model, we run 100 trainings and compare the number of times the model reaches $ge^* = 1$. The analysis is also done with and without data standardization.

The results in Table 16 show that best-found architecture provides good performance even if we use directly original traces from the DPAv4.2 and ASCADr datasets. However, with DPAv4.2, the best-found CNN architectures are less successful. For the encoded DPAv4.2 dataset, results are better than original traces as it leads to either similar performance or often better. With the dataset encoded with ae_cnn_dpav42_best_700, we got better results without standardization, and for the encoded dataset from ae_mlp_dpav42_best_700, it was better using standardization.

Using original traces was already very successful for the ASCADr dataset, so using encoded data is less valuable, but still shows good performance when the dataset is encoded with ae_cnn_ascadr_best_700, especially with standardization. On the other hand, using ae_mlp_ascadr_best_700 encoded data usually resulted in worse outcomes. Considering the standardization of encoded data, we see that it was slightly beneficial to use standardization for data encoded with both ae_mlp_ascadr_best_700 and ae_cnn_ascadr_best_700 encoded cases. Statistically, however, we cannot claim that it is always necessary to use standardization. On the other hand, based on our results, models trained with encoded data perform similarly or better in most experiments than those trained with original data. In Table 16, the cases where the performance is worse are marked in red color. Thus, we conclude that using encoded data to reuse the profiling attack architecture trained with other datasets' original traces can be done despite different feature spaces. Moreover, encoded data is beneficial when the performance with the original data is unsuccessful. Hyperparameter tuning for new datasets can be significantly reduced in that way. Again, tuning is more straightforward for autoencoders by minimizing MSE and does not require the typical attack phase in classification with GE calculations.

4.4 Transfer Learning with Profiling Models to Different Encoded Datasets

We also test the benefit of autoencoders in the context of transfer learning. Again, we have the same best profiling models for the ASCADf dataset (Table 15), and we retrain the last layer to obtain the secret key byte

Table 16: Results with DPAv4.2 and ASCADr using attack architecture trained on the ASCADf dataset. The numbers represent the number of times we reach a GE of 1 when the training is done 100 times.

Model	LM	orig.	w/o stand. ae_cnn_*_best_700	w/o stand. ae_mlp_*_best_700	with stand. ae_cnn_*_best_700	with stand. ae_mlp_*_best_700
DPAv4.2						
MLP	ID	48	100	100	100	100
	HW	100	100	100	100	100
CNN	ID	0	48	1	0	100
	HW	4	96	77	100	95
ASCADr						
MLP	ID	25	9	0	75	66
	HW	100	100	99	100	100
CNN	ID	95	100	0	100	0
	HW	99	100	4	98	0

for the new dataset. In this case, the input must be the same size since we also use the trained parameters (weights and biases). Therefore, we encoded the ASCADr and DPAv4.2 datasets for transfer learning to the profiling model input size. Additionally, we again test with and without standardization of the encoded data. Since the training is faster as we train only one layer, we have a setting where we train one by one epoch, calculating the GE after each epoch and stopping when we reach a $ge^* = 1$. The maximum number of epochs is 100. Another setting is running training for a given number of epochs, which is 100, as in all our experiments.

In the results shown in Table 17, when datasets were encoded using the **ae_cnn_*_best_700** autoencoder, we see that in all cases, we reached $ge^* = 1$. Often the necessary number of epochs is small. However, when using standardization of encoded data, we see that with DPAv4.2, we reach $ge^* = 1$ in fewer cases. Standardization for encoded ASCADr did not have much influence.

Table 18 shows the attack results when datasets are encoded with **ae_mlp_*_best_700**. Here, we see that the performance is a bit worse. However, we can still reach $ge^* = 1$ in some cases without standardization. For DPAv4.2, profiling models using the identity leakage model did not reach $ge^* = 1$. Using CNN, we see that it got close to one with $ge^* = 1.65$ and $ge^* = 1.15$, so we believe this can be corrected using, e.g., more epochs. Thus, we increased the number of epochs to 150 and got

Table 17: Results for transfer learning with datasets encoded with **ae_cnn_*_best_700**. The table shares the number of epochs that the model was trained for as well as the minimum GE with a corresponding number of traces (NT).

Model	LM	w/o stand. epochs	min GE	NT	with stand. epochs	min GE	NT
encoded DPAv4.2							
MLP	ID	74	1	2878	1	23.9	2998
		100	80.8	2809	100	38.1	2998
CNN	ID	53	1	2769	82	34	2832
		100	1	309	100	57.95	2938
MLP	HW	9	1	2738	43	53.65	2743
		100	1	71	100	77.6	2724
CNN	HW	14	1	2861	37	1	2557
		100	1	812	100	1	852
encoded ASCADr							
MLP	ID	5	1	1226	24	1	2581
	ID	100	1	459	100	25.95	80
CNN	ID	10	1	1702	11	1	2286
	ID	100	1	356	100	1	382
MLP	HW	4	1	2641	4	1	2508
	HW	100	1	1621	100	1	1643
CNN	HW	12	1	2738	9	1	2157
	HW	100	1	1636	100	1	1486

a $ge^* = 1$ within $N_{ge^*=1} = 854$ traces. Similarly, we select other specific cases that did not reach $ge^* = 1$, and we experiment with the number of epochs and training two instead of one last layer in the model to verify if with those modifications we can obtain better performance. Since using the standardization primarily led to worse results, we only experimented without standardization, changing the number of epochs and the number of layers we train.

Specifically, for the MLP and HW combination with DPAv4.2, we reach $ge^* = 1$ in 12 epochs when checking the GE after every epoch. Training for 100 epochs at once, GE gets worse (3.8). Thus, we tested with only 50 epochs and reached $N_{ge^*=1} = 1993$. A combination of MLP and the ID leakage model is the worst, with minimal GE being 67.8 in 41 epochs and 115.75 after 100 epochs. Therefore, we tested multiple modifications. We tested with training two last layers in the model, again with a different number of epochs - 100, 150, and 200.

Table 18: Results for transfer learning with datasets encoded with **ae_mlp_*_best_700**. The table shares the number of epochs the model was trained for as well as the minimum GE with a corresponding number of traces (NT).

Model	LM	w/o stand.			with stand.		
		epochs	min GE	NT	epochs	min GE	NT
encoded DPAv4.2							
MLP	ID	41	67.8	2791	53	28.35	2700
		100	115.75	2827	100	77.9	17
CNN	ID	100	1.65	2935	59	44.2	2931
		100	1.15	2635	100	86.4	2915
MLP	HW	12	1	2603	1	72.1	249
		100	3.8	2501	100	136.7	0
CNN	HW	15	1	1990	19	1	2711
		100	1	747	100	1	1741
encoded ASCADr							
MLP	ID	57	18.5	14	56	21.85	1074
		100	1.95	2799	100	28.95	2973
CNN	ID	96	1.8	2992	101	6.65	2873
		100	2.55	2969	100	21.15	2826
MLP	HW	11	1	1931	12	46.1	2090
		100	7.95	2989	100	48.75	2993
CNN	HW	25	1	2430	84	1.2	2925
		100	1.55	2976	100	6.3	2894

The lowest GE are in cases with 150 epochs training one layer where we reach $\mathtt{ge}^* = 98.8$, and training two last layers with 200 and 150 epochs reaching $\mathtt{ge}^* = 104.7$ and $\mathtt{ge}^* = 99.55$, respectively. Similarly, we do this for the ASCADr dataset. In the case of MLP and the ID leakage model, again, we tested all cases as with DPAv4.2, and the improvement happens only with training the two last layers with 200 epochs getting $\mathtt{ge}^* = 1.25$. In combination with CNN and ID, the minimal GE we get is 2.55 after 100 epochs, and when we train epoch by epoch, the minimum GE is 1.8 in epoch number 96. The results indicate that the model has the capacity to learn the new dataset. We tried adding more epochs, 150 and 200, but we did not reach better results (GE was 2.5 and 4.5, respectively). Also, with only 50 epochs, we get worse results with minimal GE of 35.45. We reached $\mathtt{ge}^* = 1.3$ by training the two last layers with 150 epochs. While not investigated, it seems that perhaps using early stopping could help in this case. Early stopping could prevent GE from increasing after a certain number of epochs. The last combination we tested is the MLP and HW, where we reach a GE of 1 when training epoch by epoch. Using 50 epochs gets us to $\mathtt{ge}^* = 1.05$, and using 150 epochs results in $\mathtt{ge}^* = 1.2$. However, we already showed that we could get $\mathtt{ge}^* = 1$ training epoch by epoch. Thus, the model can learn, and early stopping could help get $\mathtt{ge}^* = 1$.

In most cases, we could get GE close to 1. In many cases, we also see capacity in the model to learn the new dataset where early stopping could be beneficial as GE seems to deteriorate after some epochs. On the other hand, training modifications did not help reach $\mathtt{ge}^* = 1$ for the MLP and the ID leakage model for the DPAv4.2 dataset. Possibly, the autoencoder requires improvements, but we also see that the results for this case specifically were better with standardization. Including standardization with training modifications might help. Additionally, from the setup with training epoch by epoch, the minimum GE is around 50 epochs and gets larger as we train for the entire 100 epochs. Thus, early stopping might also be beneficial in this case, along with other training alternatives.

Here, we see that results using data encoded with ae_cnn_*_best_700 are better than those encoded with ae_mlp_*_best_700 with and without standardization. In both cases, standardization made performance worse, so with transfer learning, we could opt not to use standardization, at least when the reused model is trained on the original dataset and now used for encoded data. However, more exploration of this can be done as the sample might be small. Additionally, if we use transfer learning from a model trained on encoded data and then used for new encoded data, this conclusion about standardization may not be valid. Still, our experiments show significant benefits of transfer learning where tuning the profiling model for a new dataset was eliminated and training time reduced. That holds while the data is also in different feature spaces as the model is trained on original data, then transferred for encoded data of other datasets.

5 Conclusions and Future Work

In this work, we proposed autoencoders to decrease the hyperparameter tuning effort of profiling models for new datasets. Hyperparameter tuning for profiling models in SCA is a necessary but time-consuming task, and additionally, those efforts are needed for each specific dataset. Thus, we propose reusing profiling models to reduce the efforts for each new dataset by using autoencoders. The commonly used metric for autoencoders is MSE, which we showed to be positively

correlated with the SNR difference between the original and reconstructed trace. Tuning autoencoders is more effortless as the MSE metric is relevant to the goal of reconstruction. On the contrary, with the classification of intermediate values, we need to perform GE calculations to validate the performance of the profiling model. Since those calculations are computationally expensive, they are not done during training, contrary to MSE computation. We perform a random search on original and encoded data of different datasets. The results showed that the tuning was not statistically significantly better with original traces, which means that encoded data does keep relevant information for the attacks.

We tested the performance of autoencoders in three portability cases. First, we considered reusing profiling architecture trained on one encoded dataset for other encoded datasets. This approach comes close to finding a universal profiling model, where all the datasets get encoded to the same feature size using autoencoders and then attacked with the same attack architecture. The results show good performance over encoded datasets. One important note is that the autoencoders with similar architecture (number of layers and neurons) lead to better attack performances. In that way, the features in encoded data are more alike, which boosts the performance of the same profiling architecture across different datasets. In the second case, we reused profiling architecture trained on one dataset's original traces for attacking new datasets with more features. Here, we use autoencoders to decrease the number of features of the new datasets to the feature size of the dataset used to train the model. Again, training with encoded data was better or similar in performance to original traces. Thus, if actual traces do not lead to good performance, we can consider using autoencoders to encode data to fewer features to achieve better performance with lower tuning efforts than finding a new profiling model. Lastly, we utilize autoencoders to allow using transfer learning between different datasets. Here, dimensionality reduction is necessary as we also keep the trainable parameters of the model. The results show great performance with data encoded with the `ae_cnn` architecture type without standardization. In other cases, we also reach $\mathtt{ge}^* = 1$ with a bit longer training or training more layers. The benefit of transfer learning enabled by autoencoders is that we eliminate the tuning of the profiling model and, additionally, significantly reduce training time for the new dataset.

In future work, CNN autoencoder types need to be more thoroughly investigated as they are more powerful considering feature extraction than MLPs. On the other hand, we should study what is represented in the latent space of autoencoders for SCA traces. We can compare autoencoders as feature processing tools with classical approaches, such as principal component analysis (PCA). Instead of running a DL-based SCA attack on encoded data, performing classical SCA on AE-encoded data would be interesting.

A Hyperparameter Search Spaces

We execute a random search over hyperparameter search spaces for autoencoders and profiling models. This section reports hyperparameter search spaces for all of our experiments. Hyperparameter search space for MLP autoencoders in the initial experiments with metric analysis and best latent size search are in Table 19. The differences are in the number of neurons per layer for the different types of autoencoders we use. Table 20 shows search space for CNN autoencoders.

Table 19: Hyperparameter search space for autoencoder ae_mlp, ae_mlp_dcr and `ae_mlp_str_dcr` in the initial experiments for metric analysis and latent dimension search.

Hyperparameter	Values	
	ae_mlp	ae_mlp_(str_)dcr
Layers	[1, 2, 3, 4, 5, 6]	
Neurons	[20, 50, 100, 250]	[20, 40, 50, 100, 150, 200, 300, 400]
Batch size	[100, 200, 400]	
Activation	[tanh, elu, selu, sigmoid]	
Learning rate	[0.005, 0.001, 0.0001, 0.00001]	
Weight init	[random_uniform, he_uniform, glorot_uniform, random_normal, he_normal, glorot_normal]	
Optimizer	[Adam, RMSprop, SGD, Adagrad]	

The batch size, activation, learning rate, weight initialization, and optimizer are the same for all AE types.

For profiling models, the hyperparameter search space for MLP and CNN is in Table 21.

Lastly, we use autoencoders with latent size 700, so we report in Table 22 the number of layers and neurons per layer we allow. Other hyperparameters stay the same as in Table 19 and Table 20.

B Statistical Tests

Using a Friedman test, we identify that there is indeed a significant difference in the means of the groups. However, we need to find out which ones differ specifically. Thus, a post-hoc test is necessary. One such test is the Nemenyi test, and using Python packages, we obtain the results for different latent sizes in Tables 23 and 24. Latent dimensions are in rows and columns. The Nemeyi post-hoc test returns the p-values for each pairwise comparison of means. Using a significance level $\alpha = 0.05$, the pairwise latent sizes with a significant

Table 20: Hyperparameter search space for autoencoder ae_cnn.

Hyperparameter	Values
Conv. layers	[1, 2, 3, 4]
Filters	[4, 8, 16]
Kernel size	[10, 20]
Strides	[5, 10]
Pool size	[2, 4]
Pool strides	[2, 4]
Pooling type	[Avg, Max]
Batch size	[100, 200, 400]
Activation	[tanh, elu, selu, sigmoid]
Learning rate	[0.005, 0.001, 0.0001, 0.00001]
Weight init	[random_uniform, he_uniform, glorot_uniform, random_normal, he_normal, glorot_normal]
Optimizer	[Adam, RMSprop, SGD, Adagrad]

Table 21: Hyperparameter search space for profiling models. For CNN models, the convolutional layer is followed by BatchNormalization and then the pooling layer. The number of filters increases by being multiplied by the corresponding order of the layer.

Hyperparameter	MLP	CNN
FC layers	[1, 2, 3, 4, 5, 6]	[1, 2]
Neurons	[20, 40, 50, 100, 150, 200, 300, 400]	[20, 50, 100, 200]
Conv. layers	-	[1, 2, 3, 4, 5]
Filters	-	[4, 8, 12, 16]
Kernel size	-	[10, 20, 30, 40]
Strides	-	[5, 10, 15, 20]
Pool size	-	[2]
Pool strides	-	[2]
Pooling type	-	[Avg]
Batch size	[100, 200, 400]	
Activation	[elu, selu, relu]	
Learning rate	[0.005, 0.0025, 0.001, 0.0005, 0.00025, 0.0001, 0.00005, 0.000025, 0.00001]	
Weight init	[random_uniform, he_uniform, glorot_uniform, random_normal, he_normal, glorot_normal]	
Optimizer	[Adam, RMSprop, SGD, Adagrad]	

difference are **bolded**. Table 23 shows the pairwise comparison for eight latent sizes because we exclude the results for `ae_mlp_str_dcr` type as with specified hyperparameter search it did not work for latent size 500. Table 24 shows results including that AE type but excluding the latent size 500.

Table 22: Hyperparameter search space for autoencoder `ae_mlp_str_dcr` with latent space size of 700. We exclude the batch size, activation, learning rate, weight initialization, and optimizer hyperparameters as they are already shown in Tables 19 and 20.

	`DPAv4.2`	`ASCADr`
Layers	[1, 2, 3, 4, 5, 6]	
Neurons	[700, 800, 900, 1000, 1200, 1400, 1600, 1800]	[700, 800, 900, 1000, 1100, 1200, 1300]

Table 23: p-values of Nemenyi post-hoc test for without the `ae_mlp_str_dcr` model.

Latent sizes	20	40	50	100	200	250	400	500
20	1.0	0.9	0.9	0.158	**0.001**	0.263	**0.001**	0.088
40	0.9	1.0	0.9	0.552	**0.015**	0.693	**0.006**	0.403
50	0.9	0.9	1.0	0.693	**0.032**	0.834	**0.015**	0.552
100	0.158	0.552	0.693	1.0	0.763	0.9	0.623	0.9
200	**0.001**	**0.015**	**0.032**	0.763	1.0	0.623	0.9	0.9
250	0.263	0.693	0.834	0.9	0.623	1.0	0.481	0.9
400	**0.001**	**0.006**	**0.015**	0.623	0.9	0.481	1.0	0.763
500	0.088	0.403	0.552	0.9	0.9	0.9	0.763	1.0

Table 24: p-values of Nemenyi post-hoc test for with the `ae_mlp_str_dcr` model.

Latent sizes	20	40	50	100	200	250	400
20	1.0	0.0	0.783	**0.042**	**0.001**	**0.030**	**0.001**
40	0.9	1.0	0.9	0.364	**0.009**	0.296	**0.001**
50	0.783	0.9	1.0	0.647	**0.042**	0.579	**0.006**
100	**0.042**	0.364	0.647	1.0	0.783	0.9	0.438
200	**0.001**	**0.009**	**0.042**	0.783	1.0	0.851	0.9
250	**0.030**	0.296	0.579	0.9	0.851	1.0	0.511
400	**0.001**	**0.001**	**0.006**	0.438	0.9	0.511	1.0

References

1. Bank, D., Koenigstein, N., Giryes, R.: Autoencoders. arXiv preprint arXiv:2003.05991 (2020)
2. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ASCAD database. J. Cryptographic Engineering **10**(2), 163–188 (2020). DOI 10.1007/s13389-019-00220-8. URL `https://doi.org/10.1007/s13389-019-00220-8`
3. Bhasin, S., Bruneau, N., Danger, J.L., Guilley, S., Najm, Z.: Analysis and improvements of the dpa contest v4 implementation. In: International Conference on Security, Privacy, and Applied Cryptography Engineering, pp. 201–218. Springer (2014)
4. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: M. Joye, J. Quisquater (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings, *Lecture Notes in Computer Science*, vol. 3156, pp. 16–29. Springer (2004). DOI 10.1007/978-3-540-28632-5_2. URL `https://doi.org/10.1007/978-3-540-28632-5_2`

5. Bronchain, O.: Worst-case side-channel security: from evaluation of countermeasures to new designs. Ph.D. thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium (2022). URL https://hdl.handle.net/2078.1/258155
6. Bronchain, O., Hendrickx, J.M., Massart, C., Olshevsky, A., Standaert, F.: Leakage certification revisited: Bounding model errors in side-channel security evaluations. IACR Cryptol. ePrint Arch. p. 132 (2019). URL https://eprint.iacr.org/2019/132
7. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without preprocessing. In: W. Fischer, N. Homma (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10529, pp. 45–68. Springer (2017). DOI 10.1007/978-3-319-66787-4_3. URL https://doi.org/10.1007/978-3-319-66787-4_3
8. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: B.S.K. Jr., Ç.K. Koç, C. Paar (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers, *Lecture Notes in Computer Science*, vol. 2523, pp. 13–28. Springer (2002). DOI 10.1007/3-540-36400-5_3. URL https://doi.org/10.1007/3-540-36400-5_3
9. Friedman, M.: The use of ranks to avoid the assumption of normality implicit in the analysis of variance. Journal of the american statistical association **32**(200), 675–701 (1937)
10. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. science **313**(5786), 504–507 (2006)
11. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Annual International Cryptology Conference, pp. 104–113. Springer (1996)
12. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: M.J. Wiener (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, *Lecture Notes in Computer Science*, vol. 1666, pp. 388–397. Springer (1999). DOI 10.1007/3-540-48405-1_25. URL https://doi.org/10.1007/3-540-48405-1_25
13. Kwon, D., Kim, H., Hong, S.: Improving non-profiled side-channel attacks using autoencoder based preprocessing. Cryptology ePrint Archive (2020)
14. Lu, X., Zhang, C., Cao, P., Gu, D., Lu, H.: Pay attention to raw traces: A deep learning architecture for end-to-end profiling attacks. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(3), 235–274 (2021). DOI 10.46586/tches.v2021.i3.235-274. URL https://doi.org/10.46586/tches.v2021.i3.235-274
15. Masure, L., Cristiani, V., Lecomte, M., Standaert, F.: Don't learn what you already know: Grey-box modeling for profiling side-channel analysis against masking. IACR Cryptol. ePrint Arch. p. 493 (2022). URL https://eprint.iacr.org/2022/493
16. Masure, L., Dumas, C., Prouff, E.: A comprehensive study of deep learning for side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(1), 348–375 (2019). DOI 10.13154/tches.v2020.i1.348-375. URL https://tches.iacr.org/index.php/TCHES/article/view/8402
17. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Transactions on knowledge and data engineering **22**(10), 1345–1359 (2009)
18. Pawar, K., Attar, V.Z.: Assessment of autoencoder architectures for data representation. In: Deep Learning: Concepts and Architectures, pp. 101–132. Springer (2020)
19. Perin, G., Wu, L., Picek, S.: Exploring feature selection scenarios for deep learning-based side-channel analysis. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(4), 828–861 (2022). DOI 10.46586/tches.v2022.i4.828-861. URL https://doi.org/10.46586/tches.v2022.i4.828-861
20. Picek, S., Heuser, A., Perin, G., Guilley, S.: Profiled side-channel analysis in the efficient attacker framework. In: V. Grosso, T. Pöppelmann (eds.) Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 13173, pp. 44–63. Springer (2021). DOI 10.1007/978-3-030-97348-3_3. URL https://doi.org/10.1007/978-3-030-97348-3_3
21. Quinn, G.P., Keough, M.J.: Experimental design and data analysis for biologists. Cambridge university press (2002)
22. Rijsdijk, J., Wu, L., Perin, G., Picek, S.: Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(3), 677–707 (2021). DOI 10.46586/tches.v2021.i3.677-707. URL https://doi.org/10.46586/tches.v2021.i3.677-707
23. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: A. Joux (ed.) Advances in Cryptology - EUROCRYPT 2009, pp. 443–461. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
24. Weiss, K., Khoshgoftaar, T.M., Wang, D.: A survey of transfer learning. Journal of Big data **3**(1), 1–40 (2016)
25. Won, Y., Hou, X., Jap, D., Breier, J., Bhasin, S.: Back to the basics: Seamless integration of side-channel pre-processing in deep neural networks. IEEE Trans. Inf. Forensics Secur. **16**, 3215–3227 (2021). DOI 10.1109/TIFS.2021.3076928. URL https://doi.org/10.1109/TIFS.2021.3076928
26. Wu, L., Perin, G., Picek, S.: I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. IEEE Transactions on Emerging Topics in Computing pp. 1–12 (2022). DOI 10.1109/TETC.2022.3218372
27. Wu, L., Picek, S.: Remove some noise: On pre-processing of side-channel measurements with autoencoders. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 389–415 (2020). DOI 10.13154/tches.v2020.i4.389-415. URL https://tches.iacr.org/index.php/TCHES/article/view/8688
28. Yosinski, J., Clune, J., Bengio, Y., Lipson, H.: How transferable are features in deep neural networks? Advances in neural information processing systems **27** (2014)
29. Zhou, Y., Standaert, F.: Deep learning mitigates but does not annihilate the need of aligned traces and a generalized resnet model for side-channel attacks. J. Cryptogr. Eng. **10**(1), 85–95 (2020). DOI 10.1007/s13389-019-00209-3. URL https://doi.org/10.1007/s13389-019-00209-3
30. Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., He, Q.: A comprehensive survey on transfer learning. Proceedings of the IEEE **109**(1), 43–76 (2020)