Playing With Blocks: Toward Re-Usable Deep Learning Models for Side-Channel Profiled Attacks

Servio Paguada, Lejla Batina, *Senior Member, IEEE*, Ileana Buhan, and Igor Armendariz

***Abstract*— This paper introduces a deep learning modular network for side-channel analysis. Our approach features a deep learning architecture with the capability to exchange parts (modules) with other neural networks. We aim to introduce reusable trained modules into side-channel analysis instead of building architectures from scratch for each evaluation, reducing the body of work. Our experiments demonstrate that our architecture feasibly assesses a side-channel evaluation, suggesting that learning transferability is possible using the architecture we propose in this paper.**

***Index Terms*— Side-channel analysis, modular network, deep learning, autoencoders, transfer learning.**

I. Introduction

IN THE side-channel analysis (SCA) field of research, deep learning models (DL models) are powerful tools to evaluate the implementation of secure algorithms. Unfortunately, despite the significant accomplishments achieved using deep learning models, many challenges remain. One of those challenges is mitigating the difficulty in designing architectures for each evaluation's scenario.

When evaluating a secure implementation of an IoT device, for example, it is challenging to develop a deep learning classifier that feasibly assesses the resilience of the devices. Electronic noise as countermeasure and desynchronization are specific challenges during the evaluation. Indeed, a noisy signal intrinsically suggests dealing with high-dimensional signals. For instance, targeting a modern System-on-Chip with high clock frequencies requires increasing the sampling resolution. Consequently, the side-channel information needed for the evaluation contains leakage traces with several irrelevant features (sample points). Then, noise filters and feature engineering as pre-processing steps are being considered as tools to deal with those challenges [1]–[6].

This paper proposes a new technique to overcome these challenges and introduces a novel approach that uses a deep learning classifier whose part of its architecture allows it to be re-used in other models that use the same method. By featuring exchangeable modules, we can re-use networks for different SCA evaluations, reducing the body of work of deriving models each time. The suggested architecture comprises *coupled* modules. Those modules trains to deal with a specific task of the SCA evaluation. For instance, the classification task and the pre-processing task. We call this approach DL-SCA modular network. In particular, the two modules we suggest in this paper are an autoencoder and a convolution-based classifier.

An autoencoder can effectively deal with the problem of high dimensionality and the problem of noise. An autoencoder comprises two parts; the encoder and decoder. The encoder has a last layer known as embedding, where high-dimensional leakage traces are transformed into lower-dimensional leakage traces. Because of it, autoencoders are learning algorithms used in pre-processing steps *i.e.* feature extraction [2], [7]–[9].

The classifier module serves two objectives; (i) the classification required for the SCA evaluation and (ii) regularizing the autoencoder. As we explain in further sections of this paper, autoencoders might fail to compress the samples taken from the device under test. Hence, penalizing it with a regularization might correct it toward better performance.

Our experiment uses datasets with desynchronization and countermeasures. After proving the effectiveness of the DL-SCA modular network, we perform a second set of experiments where we exchange the modules between modular networks. Our results show that transferability is feasible and applicable to side-channel analysis. The contributions of this paper are as follows:

- We introduce an approach called DL-SCA modular network and deep learning architecture featuring the exchange of modules through models. We provide the implementation details of the architecture and the hyperparameters to take into account in the design to avoid pitfalls.
- We present a training strategy based on the sharing weight technique and early stopping policy for seamless adoption of our approach in current SCA evaluations.

Manuscript received 2 March 2022; revised 27 June 2022; accepted 25 July 2022. Date of publication 3 August 2022; date of current version 12 August 2022. This work was supported in part by the Ayudas Cervera para Centros Tecnológicos grant of the Spanish Centre for the Development of Industrial Technology (CDTI) through the Project EGIDA under Grant CER-20191012; and in part by the Basque Country Government through the ELKARTEK Program, Project REMEDY-Real Time Control And Embedded Security under Grant KK-2021/00091. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Nele Mentens. *(Corresponding author: Servio Paguada.)*

Servio Paguada is with the Digital Security Group, Radboud University, 6525 EC Nijmegen, The Netherlands, and also with the IKERLAN Research Center, Arrasate, 20500 Basque, Spain (e-mail: serviopaguada@gmail.com).

Lejla Batina and Ileana Buhan are with the Digital Security Group, Radboud University, 6525 EC Nijmegen, The Netherlands.

Igor Armendariz is with the IKERLAN Research Center, Arrasate, 20500 Basque, Spain.

Digital Object Identifier 10.1109/TIFS.2022.3196273

- We elaborate experiments that demonstrate the effectiveness of re-using modules through modular networks, using different "sharing" protocols based on non-trainable layers.

The rest of the paper is organized as follows: Sect. II details theoretical aspects of the topics used for this work. Related works are discussed in Sect. III. Sect. IV provides information about datasets used in the experiments. Sect. V discuss the main contribution of this paper. Sect. VI and Sect. VII discuss the experiments. While, Sect. VIII concludes the paper.

II. Background

A. Profiled Attack

A side-channel attack requires a *leakage model* to attack the sensitive information contained in a target device. A leakage model refers to a function (that we denote as δ) that models the leak of sensitive information. Using a leakage model, an adversary can steal the secret key from a device that implements a cryptographic algorithm. The equation (1) is a leakage model used to attack a cryptographic implementation of AES,[1] this leakage model is called the identity leakage model. The variable p is a plaintext and k is a secret key candidate that takes values from a keyspace $\mathcal{K} = \{0, \dots 255\}$ which correspond to 1 byte.

$$\delta = \text{S-box}(p \oplus k) \tag{1}$$

The adversary measures the power consumption[2] when the device inputs the AES algorithm with p random values taken from $\mathcal{K}$ drawing several leakage traces (a.k.a power traces). We can define a leakage trace $\mathbf{t}_i \in \mathcal{X}$, and its dimension being denoted as $dim(\mathbf{t}_i) = m$. The $\mathcal{X}$ set comprises all the leakage traces drawn from the device. With enough leakage traces, the adversary can find a correlation between the power consumption and the inputs p of the leakage model; consequently, he can infer the secret key k^* by searching through all the secret key candidates and sorting them using their likelihood.

Overall, there are two categories of side-channel attacks (non-profiled attacks and profiled attacks). We have described the common parts found in them. In this paper, we focus on profiled side-channel attacks. A profiled side-channel attack uses classifiers to distinguish the outputs of a leakage model. The attack splits into two phases; (i) to train a classifier $\mathcal{C}$ (profiling phase) and (ii) to perform the attack (attack phase). The first phase involves applying the corresponding leakage model to a clone device (a.k.a. profile device). The adversary trains a classifier using a set of profiling traces ($\mathcal{X}$) collected from the clone device. Once the classifier is ready, the adversary obtains another set of traces, but this time from the target device. The classifier process the attack traces to compute probabilities during the attack phase. Then a key recovery process takes place using an algorithm called guessing entropy, which we will briefly explain.

It is well-known that coming up with a classifier for SCA evaluation is not straightforward. Designing a classifier for each assessment is tricky as each cryptographic implementation and device requires a specific trained model. This paper's motivation is to reduce the work of building a classifier for each SCA evaluation. We propose a deep learning-based model whose architecture allows the model to exchange classifiers with other deep learning models to conduct SCA evaluation over a different device. In the following, we address the necessary aspects that support the theory of this approach.

B. Deep Learning-Based Profiled Attacks

A deep learning classifier outputs a vector of probabilities fed into the guessing entropy (GE) metric to compute the rank of the key (k^*). We denote a deep learning model $\mathcal{C}_\delta^\theta$ for profiled attacks as a classifier $\mathcal{C}$ with a vector of hyperparameters $\theta \in \mathbb{R}^n$. The classifiers aims to distinguish leakage traces labeled using a leakage model δ. Having labeled leakage traces means that our learning approach is supervised learning [10] which represents one of the most feasible ways to leverage the learning of a deep learning classifier. Despite several deep learning architectures, convolutional neural networks (CNNs) based models are the preferred architecture to use in profiled attacks. The convolutional part plays an essential role when leakage traces are desynchronized. The deep learning model we propose uses a specific type of convolution, called dilated convolution [11] for boosting the feature extraction capability of the layer (see sub-section II-E).

C. Guessing Entropy (GE)

GE is the average *rank* of the correct key byte value k^* in a key guessing vector $\mathbf{g}$, over all the set $\mathcal{K}$ of key candidates k [12]. Formally denoted as $\text{GE} = rank_{k^*}(\mathbf{g})$, where $rank_k(\mathbf{g}) \in \{0, \dots, |\mathcal{K}| - 1\}$, and the key guessing vector is defined as: $\mathbf{g} = sort(\mathbf{E}[log\mathbf{P_r}(\mathbf{t}_i; \mathcal{C}_\delta^\theta)])$. $\mathbf{P_r}(\mathbf{t}_i; \mathcal{C}_\delta^\theta)$ is the input vector of probabilities $\mathbf{p}_{i,j}$ from a classifier (usually aimed for key recovery task) given a leakage trace $\mathbf{t}_i$. After applying the expectation $\mathbf{E}$ per multiples experiments of $\mathbf{P_r}$, the *sort* function orders the resultant vector $\mathbf{g}$ in decreasing order of probabilities. The element $g_0 \in \mathbf{g}$ corresponds to the most likely key candidate, while $g_{|\mathcal{K}|-1} \in \mathbf{g}$ is the less likely one.

D. Feature Extraction

A feature extraction process applies a transformation (linear or non-linear) to a space of observations resulting in a new space mapped by the transformation. Formally, given profiling set $\mathcal{X}(N, m)$ where N is the number of leakage traces in the set, and each trace comprises m features (or sample points). Feature extraction applies a function F to the profiling set $\mathcal{X}$ mapping a new profiling set $\mathcal{Y}$ whose elements have fewer dimensions of the corresponding elements in $\mathcal{X}$; precisely, F is an application such as $F: \mathcal{X} \longmapsto \mathcal{Y}$, and $\mathcal{X} \in \mathbb{R}^m$, $\mathcal{Y} \in \mathbb{R}^n$ such that $n < m$.

This transformation aims to derive new features ($\mathcal{Y}$) from leveraging the performance of a classifier, for instance. Theoretically, features in $\mathcal{Y}$ contain the "transformed" information that best represents the ground truth of $\mathcal{X}$. In the SCA case, it is the leakage of sensitive information. Simply put, the

[1] Or any other cryptographic primitive with non-linear functions.

[2] Being the most traditional measurement used in SCA, others are Electromagnetic Emanation, heat, sound, and several more.

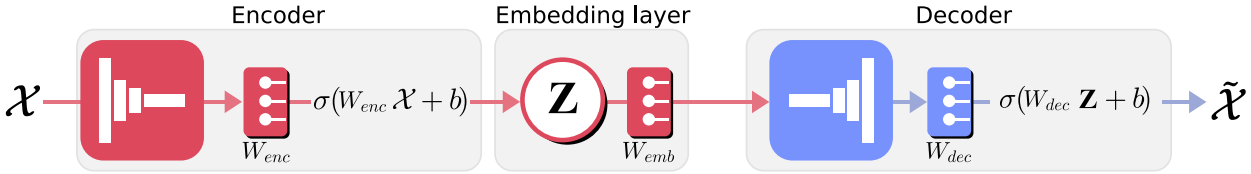

Fig. 1. Typically autoencoders are symmetrical models, meaning that both parts of the encoder and decoder resemble each other. During training, the encoder trains to code the original signal to a *Latent space*, ideally this code from the features that better represent the characteristic of the original signal. From there, the decoder re-constructs as much as possible the original signal.

intensity of the valuable information gets emphasized while the irrelevant information (non-correlated information) has little to no influence in the new space. However, developing a transformation F that certainly highlights the side-channel information is not straightforward. A transformation that goes wrong discards a lot of helpful information, and it happens when F cannot keep the variance that distinguishes a leakage trace from another. Consequently, $\mathcal{Y}$ is made of several collapsed traces becoming useless for classification purposes. In section V, we will discuss how our proposed method implements regularization to avoid transformations that collapse the $\mathcal{Y}$ space.

Function F can be inferred directly from $\mathcal{X}$. For instance, Principal Components Analysis (PCA) [13] or Linear Discriminant Analysis (LDA) [14] are two algorithms to build linear base F functions for feature extraction. However, PCA and LDA are highly sensitive to desynchronization because of their "per feature" process, meaning they find a relation by correlating the same positioning feature through samples. So that, when the samples have a spatial disruption, the relation gets reduced, requiring more samples.

E. Autoencoders

An autoencoder is a learning algorithm useful to infer F. Contrary to PCA and LDA, an autoencoder can infer a non-linear transformation due to the non-linear activation functions in its architecture. Moreover, when the autoencoder architecture consists of convolution layers, it handles the spatial disruption better than PCA and LDA.

An autoencoder consists of two parts; (i) an encoder φ and (ii) a decoder ψ. The encoder outputs a new trace $\mathbf{t'}_i$ with $dim(\mathbf{t'}_i) < dim(\mathbf{t}_i)$ (see expression (2)). At the other side of the autoencoder, the decoder tries to reconstruct $\mathbf{t}_i$ but it is able to re-build an approximation $\tilde{\mathbf{t}}_i$ only; consequently, one can understand that an autoencoder learns by minimizing the difference between $\mathbf{t}_i$ and $\tilde{\mathbf{t}}_i$ (as we will see in expression 5).

$$\mathbf{t'}_i = \varphi(\mathbf{t}_i), \quad \tilde{\mathbf{t}}_i = \psi(\mathbf{t'}_i) \tag{2}$$

From a functional perspective, the encoder maps $\mathcal{X}$ to an embedding space denoted by $\mathbf{Z}$ (i.e. $\varphi\colon \mathcal{X} \mapsto \mathbf{Z}$), the embedding $\mathbf{Z}$ is usually called latent space, code, latent code or hidden code. Further, $\mathbf{Z}$ is the space result of the transformation applied by the encoder. According to the discussion in the previous sub-section, $\mathbf{Z}$ is the resultant space of a feature extraction process *i.e.* $\mathcal{Y}$. Likewise, the decoder maps $\mathbf{Z}$ to $\tilde{\mathcal{X}}$ (*i.e.* $\psi : \mathbf{Z} \mapsto \tilde{\mathcal{X}}$), where $\tilde{\mathbf{t}} \in \tilde{\mathcal{X}}$. The expressions (3a) and (3b) formalize these two mappings;

$$\mathbf{Z} = \varphi(\mathcal{X}) = \sigma(W_{enc}\,\mathcal{X} + b) \tag{3a}$$
$$\tilde{\mathcal{X}} = \psi(\mathbf{Z}) = \sigma(W_{dec}\,\mathbf{Z} + b') \tag{3b}$$

Function σ denoted a non-linear activation function. An encoder is parameterized by a weight matrix $W_{enc} \in \mathbb{R}^{m\times n}$ and a bias vector $b \in \mathbb{R}^n$; likewise, a decoder is parameterized by a weight matrix $W_{dec} \in \mathbb{R}^{n\times m}$ and a bias vector $b' \in \mathbb{R}^m$ (see Fig. 1). Training an autoencoder implies finding a vector of parameters $\theta = (W_{enc}, W_{dec}, b, b')$ that minimize a loss function $\mathcal{L}$ such as;

$$\Theta = \min_{\theta} \mathcal{L}(\mathbf{t}, \tilde{\mathbf{t}}) = \min_{\theta} \mathcal{L}(\mathbf{t}, \psi(\varphi(\mathbf{t}))) \tag{4}$$

As we said, autoencoder learns by minimizing the difference between $\mathbf{t}_i$ and $\tilde{\mathbf{t}}_i$; so that, the Mean Square Error (MSE) is a loss function commonly used;

$$\mathcal{L}_{\text{MSE}} = \mathcal{L}(\mathbf{t}, \tilde{\mathbf{t}}) = \frac{1}{m}\sum_{i=1}^{m} (\mathbf{t}[i] - \tilde{\mathbf{t}}[i])^2 \tag{5}$$

1) Convolution Layer Architecture: Autoencoders are built using either fully connected layers or convolutional layers. The latter makes the autoencoder inherit the spatial invariant robustness property, which is useful when leakage traces are desynchronized; our autoencoder uses dilated convolution layers.

A convolution layer consists of kernels that essentially are matrices; then, to dilate a convolution layer consists of inserting zeros into its kernels, meaning to separate the matrices' elements using zeros, expanding their receptive field.[3] According to [11] a dilated kernel allows convolutions base classifiers to combine spread features that contain the leakage information, at the same time, avoiding irrelevant features that might lay in between.

Let us consider the expression in (6) showing a regular convolution where a leakage trace $\mathbf{t}_i$ is multiplied by the kernel $\mathbf{q}$ whose length is denoted by $l_{\mathbf{q}}$. If we displace the leakage trace $\mathbf{t}_i$ from right to left, a single feature t of $\mathbf{t}_i$ is multiplied $l_{\mathbf{q}}$ times. If $l_{\mathbf{q}}$ is large, then t might be *excessively* used during the operation. According to [15], this excessive use of t may decrease the convolution effectiveness. Notice that if $l_{\mathbf{q}}$ increases aiming to use further spread features, it also increases the times t is used. By using dilated convolutions,

[3]Called it to those kernel elements which are not zero.

one can avoid this downside.

$$\begin{aligned}(\mathbf{t} \circledast \mathbf{q})[t] &= \sum_{n=-\infty}^{\infty} \mathbf{t}[t-n] \cdot \mathbf{q}[t] \\ &= \cdots \\ &\quad + \underbrace{(\mathbf{t}[t-n_i] \cdot \mathbf{q}[n_i]) + (\mathbf{t}[t-n_{i+1}] \cdot \mathbf{q}[n_{i+1}])}_{l_{\mathbf{q}} \text{ times}} \\ &\quad + \cdots \end{aligned} \tag{6}$$

The expression in (7) shows a dilated kernel with one zero inserted between its elements. Notice that when the convolution is performed, the feature t alternates being multiplied or not by a zero; consequently, it reduces the times the operation uses the feature t.

$$\begin{aligned}(\mathbf{t} \circledast \mathbf{q}_d)[t] &= \sum_{n=-\infty}^{\infty} \mathbf{t}[t-n] \cdot \mathbf{q}_d[t] \\ &= \cdots + \\ &\quad \left.\begin{array}{l}(\mathbf{t}[t-n_i] \cdot 0) + \\ (\mathbf{t}[t-n_{i+1}] \cdot \mathbf{q}_d[n_{i+1}]) + \\ (\mathbf{t}[t-n_{i+2}] \cdot 0) + \\ (\mathbf{t}[t-n_{i+3}] \cdot \mathbf{q}_d[n_{i+3}])\end{array}\right\} \hat{l_{\mathbf{q}}} \text{ times} \\ &\quad + \cdots \end{aligned} \tag{7}$$

The hyperparameter *dilatation rate* (dr) controls the number of zeros inserted. When a kernel is dilated, its receptive field is modified by the relation.

$$\hat{l_{\mathbf{q}}} = l_{\mathbf{q}} + (l_{\mathbf{q}} - 1)(dr - 1) \tag{8}$$

The kernel's receptive field consists of those elements that are active, in other words, elements that are non-zero. In this way, the receptive field increases by modifying either the length of the kernel or the dilatation rate, letting the user regularize the convolution operation.

III. Related Work

While few works in SCA discuss an approach of architecture transferability with reusable modules, several works have discussed feature reduction for SCA. Cagli *et al.* in [16]–[18] discussed application of traditional feature reduction methods using PCA [13], LDA [14], and its kernel base variant Kernel PCA and KDA. Picek *et al*. [19] published results using same methods as [17]. However, authors in [19] used an approach that combined feature extraction and feature selection; precisely, PCA and LDA combined with SOST and SOSD, they called it hybrid feature selection methods.

Intrinsically, any work that uses the same feature reduction techniques aims to downsample the signal by taking it to a new space (latent space). However, these approaches consider only linear base feature reduction disregarding the more powerful non-linear version of it; it is likely, that this situation may be a consequence of advertising CNNs as built-in feature extraction deep learning models. Hence, very few works have addressed non-linear methods for SCA evaluation. One of those few works are, for instance, Paguada *et al*. [2], and Yang *et al*. [20]; similar to us, those works used autoencoders toward inferring a non-linear function to pre-process leakage traces in a fashion that overcome linear methods.

TABLE I

Cardinalities of the ASCAD Datasets. Since Their Goal Is to Be Used for Benchmarking Profiled Attacks the Leakage Traces Are Grouped in profiling_traces and the attack_traces Sets

ASCAD^f		**ASCAD^r**	
Profiling_traces	50 000	Profiling_traces	200 000
Attack_traces	10 000	Attack_traces	100 000
dim($\mathbf{t}_i$)	700	dim($\mathbf{t}_i$)	1 400

While those two works are the most related to use, we state that, to the best of our knowledge, no previous work on side-channel analysis suggests a deep learning approach with the capability to share modules through different neural networks.

IV. ASCAD Fixed and Random Datasets

ASCAD dataset[4] was introduced in [21]. The leakage traces were collected from an Atmega8515 8-bit microcontroller. The cryptographic algorithm implemented is AES-128 protected using masking countermeasure [22], [23].

The dataset has two versions, traces collected with fixed key encryption k_f and traces collected with random key encryption k_r (plaintext is always random), while the target byte of the secret key in both cases is the third one. We named these versions as ASCAD^f and ASCAD^r respectively. Due to these key characteristics, ASCAD^r is more challenging and more realistic than ASCAD^f when conducting an SCA evaluation over them. TABLE I contains a summary of main characteristics of these two datasets.

Leakage traces in each version are desynchronized according to a threshold value that moves traces around the x-axis, being frequently used threshold values of 0, 50, and 100. Then, to make clear distinctions when exchanging the modules between modular networks, we add to the name the threshold value, for instance, ASCAD^r $desync50$.

V. DL-SCA Modular Network Architecture

This section explains the details of the architecture of the DL-SCA modular network; further, we describe the strategy to train it.

Since we are using autoencoders our suggested DL-SCA modular network comprises three main modules; an encoder, a decoder, and a classifier (see Fig. 2). We group the encoder and decoder into a single module called a *downsampler*. The downsampler has two goals; (i) to extract meaningful features by reducing the noise in the leakage traces and (ii) to downsample them. Now, the classifier is in charge of evaluating those extracted features as a classification problem. It is worth mentioning that once the DL-SCA modular network is trained, we discard the decoder of the downsampler, and we only use the encoder and classifier to perform the SCA evaluation. Due to this, we elaborate a *training strategy* to monitor only those two parts of the model; we will elaborate this later in this section.

[4]This dataset is publicly available at https://github.com/ANSSI-FR/ASCAD

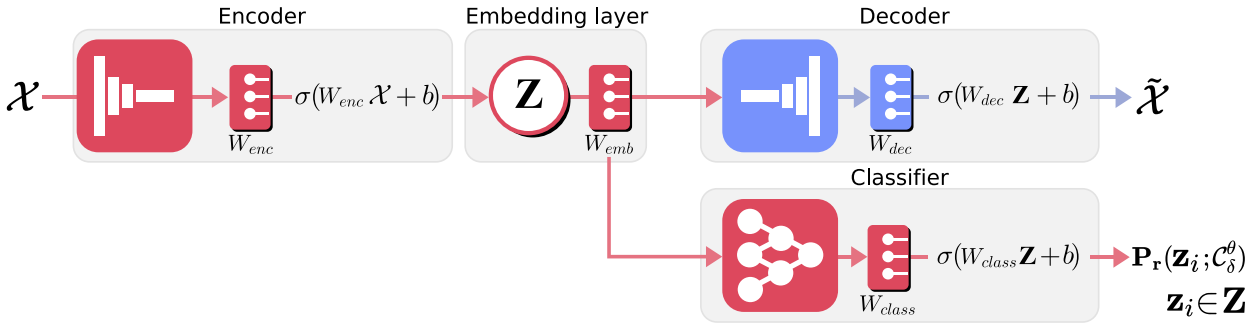

Fig. 2. DL-SCA modular network architecture illustration. The encoder, with its embedding layer and the classifier ensemble the final model used to perform the attack.

To achieve compatibility with as many classifiers as possible, we should use a downsampler to fix the classifier input. Precisely, we downsample the leakage traces to a fixed length. Then, when we re-use the classifier with another downsampler, this latter fixes its output to match the classifier's input. By doing this, we fulfill the first step of re-usability. We demonstrate this in the experimental section of this paper.

Training a DL-SCA modular network architecture requires a loss function for the decoder and another for the classifier. The decoder's loss function ($\mathcal{L}_{MSE}$) was discussed in the sub-section II-E. We introduce the classifier loss function.

A. Classifier Loss Function

A classifier outputs a vector of probabilities used as input for the guessing entropy. To output this vector of probabilities, it must be trained using a cross-entropy (CE) loss function. In supervised profiled side-channel attacks, the leakage traces are labeled by the output of a leakage model (see expression (1)). Further, the classifier learns by minimizing its error in predicting the label of each trace. To explain this better, let us consider the expression (9). The space $\hat{\mathcal{K}}$ corresponds to a batch of key candidates or labels, each one of the labels in $\hat{\mathcal{K}}$ corresponds to a trace. For instance, let us take $\delta_i \in \hat{\mathcal{K}}$ as one of those labels, we say that δ_i is the ground truth while $\sigma(\delta_i)$ is the output score a neural network computed.[5]

$$\mathcal{L}_{CE} = -\sum_{i}^{\hat{\mathcal{K}}} \delta_i \cdot \log(\sigma(\delta_i)) \tag{9}$$

During training, this loss function computes the error in the prediction made by the classifier. Consequently, the weights of the classifier are updated toward achieving a prediction with the highest accuracy possible.

Clearly, we use a classifier with the same purpose as in a common profiled side-channel evaluation. However, the additional purpose in using a classifier in our approach is to add a regularization term to the downsampler. Precisely, the supervised classifier adds an extra penalization to the downsampler with regard to the feature space $\mathcal{Y}$ leading the whole network toward better performance. The arrangement depicted in Fig. 2 shows how both the classifier and decoder attach to the encoder. When training the modular network, the classifier feed-forwards the downsampled traces from the embedding and back-forwards its loss which penalizes the encoder. Meanwhile, the decoder trains its reconstruction capability that additionally penalizes the encoder. These two losses resemble a double voting system that the encoder uses to leverage learning.

Now, notice that because the activation functions are non-linear, the classifier acts as a non-linear regularizer for the embedding space. The decoder takes the regularization effect as small perturbations in that space. Those perturbations challenge the decoder in reconstructing the original traces as it understands that those are small errors in its reconstruction. Contrary to the approach in [2], training jointly the autoencoder and the classifier produces an embedding likely to learn relevant features. Due to the regularization factor, less correlated features are emphasized over highly correlated noisy features.

B. Analogy With Linear Regularized Autoencoder

Autoencoders aim to be imperfect models. Hence, when training an autoencoder, we must avoid an architecture that ends with a model called "identity function". When this phenomenon happens, the autoencoder will copy the data from the input to the output. One way to avoid this is to use an under-complete architecture, which refers to the embedding we discussed earlier. Another workaround is to build deep autoencoders; the deeper an autoencoder is, the stronger it becomes to avoid ending as an identity function. However, building a deep autoencoder carelessly might reduce the network performance as the model becomes too complex for the input data. Hence, we cannot rely on it repeatedly.

Applying a regularizer to the latent space is another alternative. Regularized autoencoder proved to overcome regular autoencoders when leveraging meaningful features in the embedding. A linear regularizer applies to the latent space an extra penalization. The embedding neurons fire the additional penalization to the decoder added to its loss function as small epsilons of error. The decoder advocates the disruption by training its neurons to reconstruct the original data, ignoring that it is being fooled by the regularizer, so its learning is actually "imperfect" [10].

[5]Often σ is the softmax activation function for multi-class classification.

A drawback of using a linear regularizer is precisely its nature. A linear regularizer applies the penalization linearly to all the embedding neurons. No criterion controls the magnitude each embedding neuron should receive based on its contribution to the decoder loss function. Eventually, the decoder starts copying the input, becoming an identity function.

In a DL-SCA modular network, the classifier acts as a regularizer. Nonetheless, the regularization is based on non-linearity since the classifier is a non-linear function. The non-linear activation functions used in the classifier receive their input from the embedding neurons. Once the classifier does the back-propagation, it applies an epsilon value according to the embedding neurons' contribution to the classification. Once again, the decoder interprets those as small errors but now facing a more advanced regularization.

Both linear and non-linear regularizations require a value to control the intensity of the penalization. For our non-linear regularizer, this value is a parameter $\gamma \in [0, 1] \subset \mathbb{R}$ multiplied by the loss function's result. A zero gamma value will cancel the classifier's influence on the embedded space. In contrast, a large value (bigger than 1) will magnify the influence of the classifier. It will disable the reconstruction as this latter could not deal with the classifier's penalization added to the embedding.

C. DL-SCA Modular Network Loss Function

Now that we know the two losses required by our architecture as well as the hyperparameter to control their intensity, we have the expression (10) that defines the loss function for a DL-SCA modular network architecture.

$$\mathcal{L}_{\text{DL-SCA modular network}} = \gamma \cdot \mathcal{L}_{CE} + \omega \cdot \mathcal{L}_{\text{MSE}} \tag{10}$$

Notice that there is an ω parameter for $\mathcal{L}_{\text{MSE}}$ that works exactly as γ. We fix $\omega = 1$, because our goal is to control the regularization and not the reconstruction.

D. Training Strategy for a DL-SCA Modular Network

Recently, authors from [24] published an early stopping framework to monitor the state of a deep learning model during its training preventing it from getting overfit/underfit. Overfitting/underfitting is a phenomenon that might happen during training. It represents the state when a deep learning network cannot generalize beyond its training set.

The framework computes the guessing entropy at the end of each epoch, basing the stopping criterion on the whole guessing entropy vector. The framework considers when the guessing entropy converges and how many traces keep the guessing entropy in the state of convergence (more details can be found in the original paper [24]). We use this early stopping to elaborate a training strategy for our DL-SCA modular network.

1) Training Strategy: We know that an early stopping framework stops the training of a deep learning model when it meets conditions established using a metric, *e.g.*, the model's accuracy. Typically, these frameworks evaluate the entire model. In contrast, we need the framework to consider just the encoder and the classifier as they are the parts used in the SCA evaluation. The framework from [24] is a "typical" framework, so it monitors the whole deep learning model which is not helpful for our architecture.

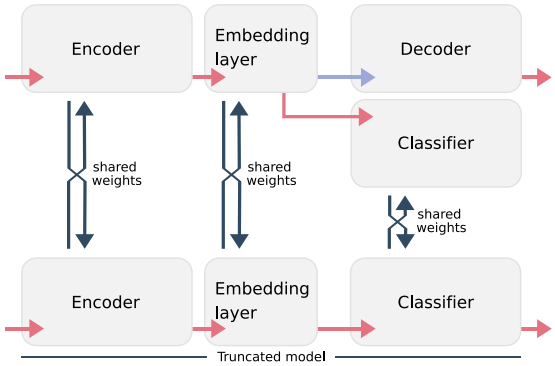

Fig. 3. The truncated model shares weights with the DL-SCA modular network; while the latter is training, the former updates his weights. The early stopping framework uses the truncated model to compute the guessing entropy at the end of each epoch, and it stops the training when it meets the conditions.

We modified the suggested framework to receive a *truncated model* which comprises just the modules of interest (encoder and classifier). To apply this modification is effortless when using the weight sharing technique [25]. In this technique, two or more neural networks share references to some specific layers. All those networks can update the weights of those layers. In our case, the original networks update the weights, while the truncated model monitors the state (of those weights) of the encoder and classifier using the early stopping framework. We set a truncated model (see Fig. 3) to reference those modules and to be only evaluated (not trained) by the early stopping framework. Then, the training process stops when the encoder generates features that make the classifier achieve the expected performance. In other words, a guessing entropy performance that converges to zero. The modification works since the framework uses the truncated model as the predictor, and its output serves as the input to compute the guessing entropy.

In the experimental results section of this paper, we show the *training strategy outcome* using surface plots. Notice that we did not stop the network training, so the surface plots correspond to the entire training process. Our goal is to show that an SCA-DL modular network does not rely on an early stopping framework.

VI. Experimental Results of Training Modules

In this section, we discuss the results of using our proposed approach over ASCAD datasets — ASCAD^f *all desync* and ASCAD^r *all desync*.

We organize the experimental results as two different use cases where a DL-SCA modular network analyzes; (i) ASCAD fixed key dataset and (ii) ASCAD random key dataset. We accomplish two goals with these uses cases; (i) to show the feasibility when an SCA evaluation uses our architecture to attack a specific dataset, and (ii) to create a scenario where we

TABLE II
DL-SCA MODULAR NETWORK ARCHITECTURE TO USE IN EXPERIMENTS WITH ASCAD^f ALL DESYNCHRONIZATIONS LEVELS

ENCODER	
Layer type	**Details**
1. Conv	# kernels: 32, kernel size: 64, SeLU dr: 3, kernel init: *He_Uniform*
2. Batch Norm	
3. Pooling	Average, # kernels: 2, stride: 2
4. Conv	# kernels: 64, kernel size: 25, SeLU kernel init: *He_Uniform*
5. Batch Norm	
6. Pooling	Average, # kernels: 25, stride: 25
7. Conv	# kernels: 128, kernel size: 3, SeLU kernel init: *He_Uniform*
8. Batch Norm	
9. Pooling	Average, # kernels: 5, stride: 5
10. Flatten	
11. Dense	300 Units (Latent space)
DECODER	
Layer type	**Details**
1. ConvTranspose	# kernels: 128, kernel size: 3, SeLU stride: 7, kernel init: *He_Uniform*
2. Batch Norm	
3. ConvTranspose	# kernels: 64, kernel size: 25, SeLU stride: 25, kernel init: *He_Uniform*
4. Batch Norm	
5. ConvTranspose	# kernels: 32, kernel size: 64, SeLU stride: 2, kernel init: *He_Uniform*
6. Batch Norm	
7. ConvTranspose	# kernels: 1, kernel size: 1, Sigmoid stride: 1, kernel init: *He_Uniform*
CLASSIFIER	
Layer type	**Details**
1. Conv	# kernels: 4, kernel size: 1, SeLU dr: 3, kernel init: *He_Uniform*
2. Batch Norm	
3. Pooling	Average, # kernels: 2, stride: 2
4. Flatten	
5. Dense	10 (units), SeLU
6. Dense	10 (units), SeLU
7. Dense	10 (units), SeLU
8. Dense	256 (units), Softmax

demonstrate the feasibility of sharing modules. The strategy is applicable to real evaluations; the derived modular network evaluates a first dataset; consequently, a second modular network could evaluate another dataset borrowing a module from a previous modular network. In our case, our experiments use two datasets that share the same source of data; precisely, both datasets were composed with leakage traces from the same microcontroller (Atmega8515 8-bit). We aim for performing experiments when the source of data is uncommon between both datasets as future works. Notice, we used the same model for all levels of desynchronization, meaning that additional effort in finding neural network architectures for specific noisy scenarios is not required.

A. ASCAD^f all desync Use Case

The TABLE II summarizes the hyperparameters of the modular network architecture to evaluate ASCAD^f $alldesync$.

1) Network's Architecture: We set the architecture by following the discussion in Sect. II-E; the first convolutional block uses dilated convolutions to avoid any useless features that might reduce the model's performance.[6] We dilate the convolutions at the first convolutional block because it is where we deal with the original version of the trace. Further, we add convolutional blocks to the encoder following the rules applied for VGG [26] base deep learning architectures.[7]

The decoder mirrors the encoder, as our downsampler uses symmetric autoencoders. For the decoder to up-sample, namely to reconstruct the actual length of the trace, it uses transpose convolutions. As known, matrix multiplication is not commutative, and we cannot achieve the same output in respective convolutional blocks. Consequently, we have to tune the hyperparameters in the decoder's convolution layers. For instance, let us take the third encoder's convolutional block that uses a stride value of 5, its corresponding decoder's transpose convolutional block is the first one but it uses stride value of 7. By doing this, we fix the output of the decoder to meet the original trace dimension.

2) Latent Space Hyperparameters: With regard to latent space units and γ value. We perform a grid search for the best number of units in the latent space, using the values 100, 200, 300, 400, and 560. We know that the parameter γ relates strictly to the number of latent units. Consequently, to find the value of γ we create combinations using the latent space values and values of γ as $\{1e^{-3}, 1e^{-6}, 1e^{-9}\}$. The best combinations was 300, and $1e^{-3}$ for latent space units and the γ parameter, respectively. Regarding the classifier module, we are only interested in its classification performance and not too much in its ability to filter out unnecessary features of the leakage traces, so we use a shallow architecture since it will deal with already filtered features.

3) Training Strategy and Results: To train a modular network, we use the early stopping framework from [24]. To show that our suggested architecture does not rely on the framework, we did not stop the training after the mentioned framework finds the best learning state. Further, we will use this outcome in the next section to discuss the result of the reusing modules experiment. Fig. 4 depicts the training process when our modular network evaluates ASCAD^f datasets. As we expected, the training outcome differs according to the level of desynchronization; regardless, our modular network achieved a zero convergent guessing entropy for all desynchronization levels. A view of the attack performance is depicted in Fig. 5.

B. ASCAD^r all desync Use Case

1) Network's Architecture: Regarding this dataset, our strategy was to keep the same modular network as the previous use case to reuse as much as possible an already worked model and see how it performs. After experimenting, we noticed that the downsampler module required an additional convolutional block —identical to the third convolutional block of the

[6]Interested readers can look at [2] for a comparison between normal convolution and dilated convolution applied to autoencoders.

[7]VGG base architectures increase their number of kernels in convolutional layers by the power of 2.

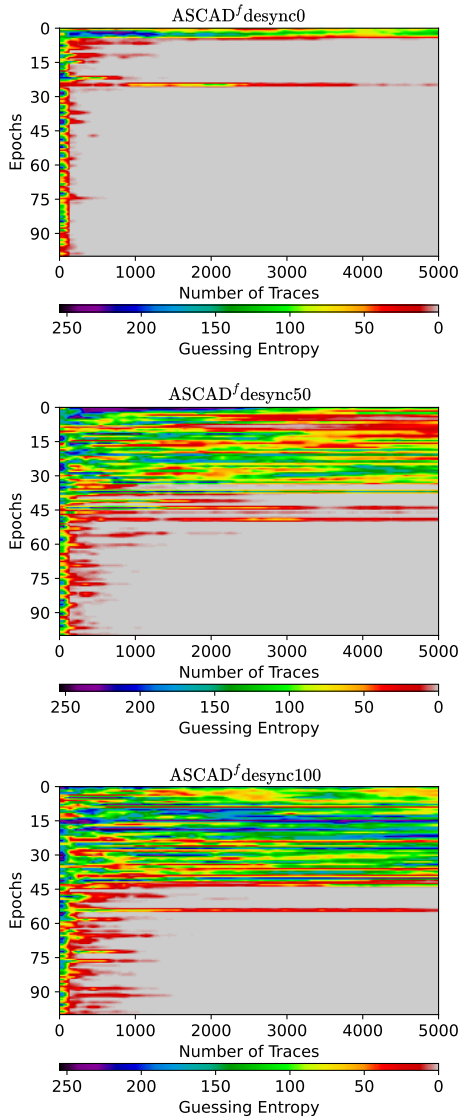

Fig. 4. The training process of the modular network for ASCAD^f datasets. The surface represents the values of the guessing entropy during a chosen number of epochs. Stopping condition success GE=0.

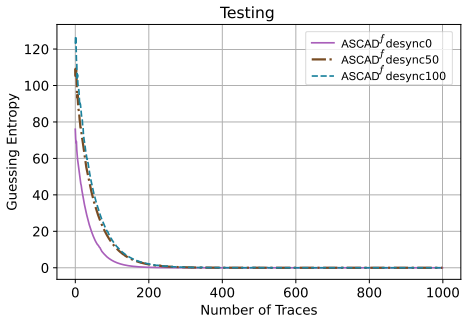

Fig. 5. Guessing entropy over ASCAD^f all levels of desynchronization.

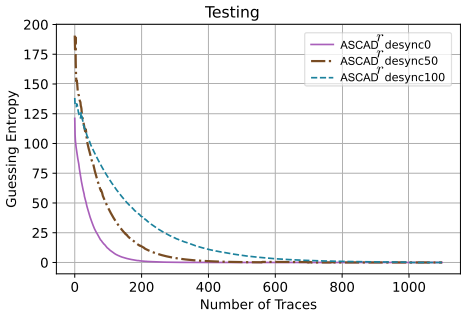

Fig. 6. Guessing entropy over ASCAD^r all levels of desynchronization.

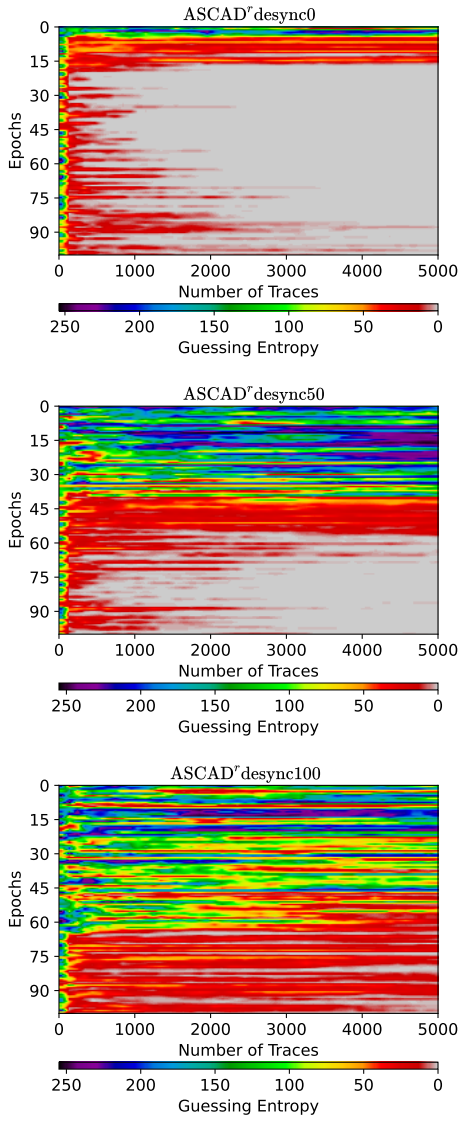

Fig. 7. The training process of the modular network for ASCAD^r datasets. The surface represents the values of the guessing entropy during a chosen number of epochs.

decoder— without pooling layer. Consequently, the decoder should also have the corresponding transpose convolutional block.

2) Latent Space Hyperparameters, Training Strategy, and Results: We keep the same classifier as in the previous use case because we have the same number of latent units. In our particular case, to keep the same latent units is convenient because we aim for exchanging a trained classifier in the following experiments to evaluate the modular re-usability. Fig. 7 depicts the training process of guessing entropy by epochs for ASCAD^r dataset. In this case, we observe that the performance of our modular network slightly decreases,

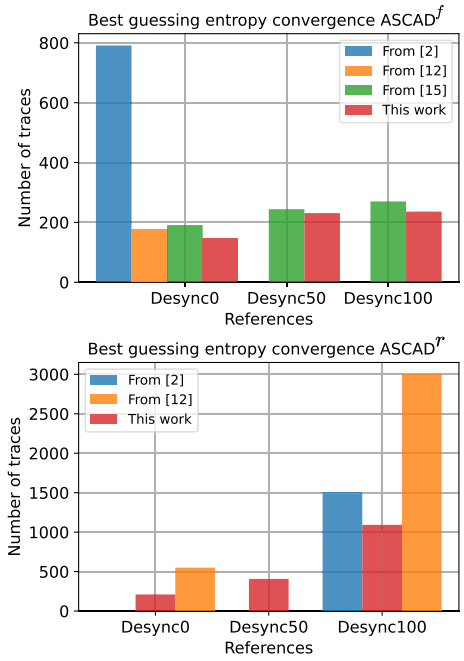

Fig. 8. Best guessing entropy convergence comparison. Some references do not report result from all the ASCAD desynchronizations.

which is expected since the dataset has a higher level of noise than the previous. Even though, we achieve good guessing entropy convergence as depicts Fig. 6. Finally, we compare our experimental results with previously reported results over the same datasets. Fig. 8 gathers this information.

VII. Module Re-Usability Experimental Results

This section presents the results of module re-usability. We show that another non-trained DL-SCA modular network can reuse the modules of a DL-SCA modular network. We use the DL-SCA-based module networks trained in the previous section.

A. Analyzing Transferability

We aim to show how "transferable" is the knowledge of a classifier module. We have six modular networks —meaning six classifiers— trained with three different datasets —three on ASCAD^f and three on ASCAD^r. Further, due to the number of latent units (300) we used, all classifiers are interchangeable without performing additional downsampling operations to fix their inputs. For our experiments, we took the classifier from the DL-SCA modular network of ASCAD^f *desync*50 to share with all the downsampler from ASCAD^r. We considered it sufficient for proving our claim about "module re-usability". We chose ASCAD^f $\longmapsto$ ASCAD^r direction because it represents the complex direction —from fixed key to random key.

We inspect the transferability of the ASCAD^f *desync*50 classifier by conducting a similarity analysis using gradient activation operations. In particular, we use heatmaps and gradient visualization to compare how the neurons' of the classifier are activated by the data outputted from the downsamplers.

TABLE III

Summary of the Similarity Analysis Between ASCAD^f *desync*50 Classifier and ASCAD^r *all desync* Classifiers

Shared classifier	Original downsampler & classifier	Gradient operation
ASCAD^f *desync*50	ASCAD^r *desync*0	heatmap
		gradient vis
	ASCAD^r *desync*50	heatmap
		gradient vis
	ASCAD^r *desync*100	heatmap
		gradient vis

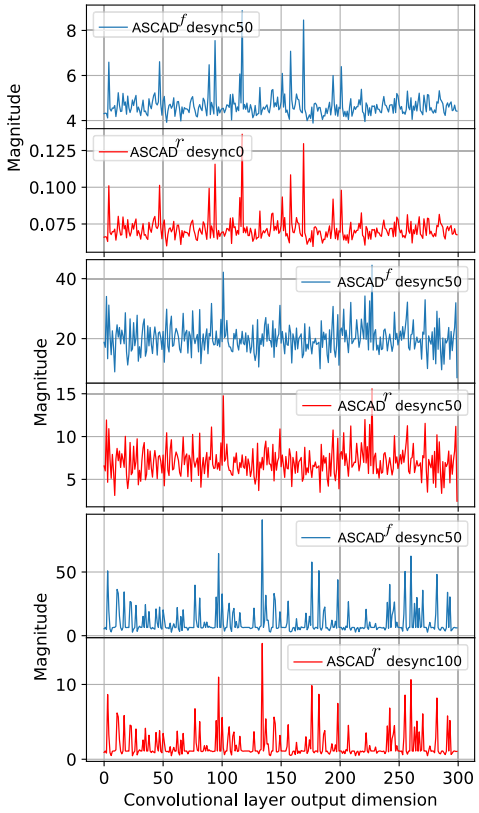

Fig. 9. Comparison between heatmaps of the ASCAD^f *desync*50 classifier and classifiers from all the ASCAD^r datasets. Notice how ASCAD^f *desync*50 heatmap resembles all other heatmaps. It indicates that ASCAD^f *desync*50 classifier's convolutional layer fires its neurons according the data received.

We perform this analysis by locking specific layers of the classifier to identify how transferable those layers are. Precisely, we choose convolutional block (Conv) layers and fully connected block (FC) layers and lock them by turns to evaluate them separately. A heatmap allows us to inspect the convolutional layers of the classifier, while gradient visualization helps us analyze how both Conv and FC perform with the different datasets.

TABLE III summarizes the similarity analysis we are going to perform using the classifier ASCAD^f *desync*50, the ASCAD^r datasets, and the gradient activation operations. Fig. 9 depicts the first convolutional layer heatmaps from ASCAD^f *desync*50 classifier and ASCAD^r *all desync* classifiers (*desync*0, *desync*50, and *desync*100).

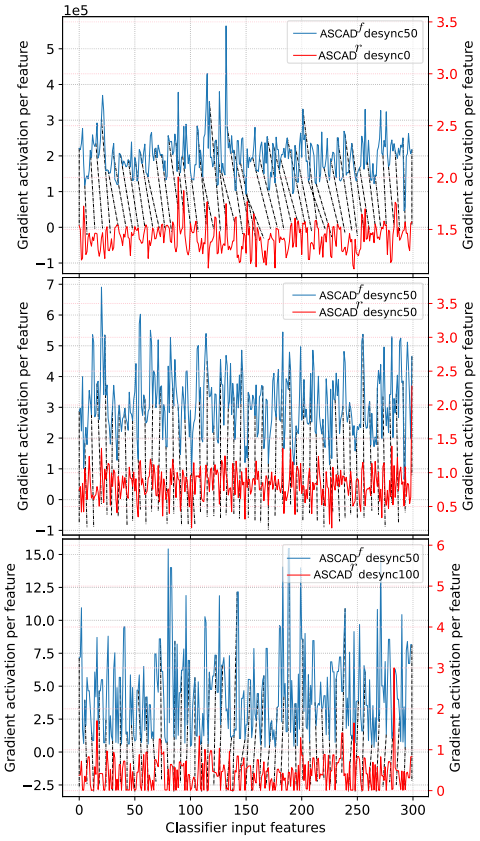

Fig. 10. Comparison between gradient activation per sample of the ASCAD^f $desync50$ classifier and classifiers from all the ASCAD^r $all desync$.

For these particular experiments, all ASCAD^r classifiers share similarities with the ASCAD^f $desync50$ classifier in how their convolutional layer neurons' get stimulated. According to our assumptions, it indicates that the weights of those layers might be transferable. This claim is experimentally demonstrated later in the final experiments.

Although the magnitude of the ASCAD^f $desync50$ classifier's heatmap is higher than any other heatmap from ASCAD^r classifiers, it does not represent a drawback to the transferability. We could have gotten the same magnitudes if we had normalized the weights applying constraints in the architecture. The important aspect of the heatmap is to show from which points of the leakage trace (in this case, a compressed leakage trace) a distinguisher influences its classification. As shown in Fig. 9, the points in the respective plots (*i.e.* ASCAD^f desync50 and ASCAD^r desync0, ASCAD^f desync50 and ASCAD^r desync50, and so on) have the highest magnitude in the same corresponding points.

We use gradient visualization to inspect the classifiers' fully connected block (FC). The output of that operation indicates which input *features* are the most *meaningful* for the classification. The gradient visualization uses the loss function of a trained classifier to conduct backpropagation, collecting the information about those neurons that emphasize the performance. Further, when it reaches the input layer, it points out which features are connected to those neurons, indicating the meaningful features [27]–[29]. Fig. 10 depicts the result of gradient visualization operation.

TABLE IV
COMBINATION OF SHARING PROTOCOLS USED FOR THE ASCAD^f $desync50$ CLASSIFIER

Classifier		Re-used in
Shared classifier	**Lock use case**	**Dataset & filter**
ASCAD^f $desync50$	Convlock	ASCAD^r $desync0$
	Bothlock	
	FClock	
	Convlock	ASCAD^r $desync50$
	Bothlock	
	FClock	
	Convlock	ASCAD^r $desync100$
	Bothlock	
	FClock	

Notice that gradient visualization shows less intuition than heatmap. As a workaround, we apply a Dynamic Time Warping (DTW) [30] to visualize the similarities between gradient visualization signals.

According to this experiment, two phenomena happen; (i) the meaningful features are displaced according to each classifier, or/and (ii) the meaningful features are less intense in magnitude. These phenomena could represent an issue. For instance, let us take the ASCAD^r $desync0$ classifier, notice the displacement because the ASCAD^f $desync50$ interprets that the meaningful features localize differently. Further, those features have an even lower magnitude in contrast to those supposedly being the lowest (see points from 0 to 30 in Fig. 10 top plot).

This analysis gives us the intuition that we will need to retrain the classifier. Nevertheless, the reader might remember that the classifier is just a part of a bigger model. The downsampler will leverage its learning according to the limitation imposed by the classifier.

B. Playing With Blocks

Let us suppose we have trained a DL-SCA modular network using a former dataset; then, we have the opportunity to evaluate another dataset. We could use the classifier module of the first network to evaluate it. In this hypothetical scenario, the first dataset is played by ASCAD^f and the second one by the ASCAD^r dataset.

To experimentally evaluate if we need to re-train some or all parts of the classifier, we perform experiments locking the blocks of the classifier to restrict them from getting trained. In the previous sub-section, we inspected the blocks of the classifier (Conv and FC), and we observed some similarities in its neurons' weights. Now, we are going to evaluate the performance of the whole modular network when its classifier module has the following locks:

- Convolutional block
- Fully-connected block
- Both blocks

We will refer to these as "sharing protocols". We find out which could be the best sharing protocol for these particular modular networks by locking the blocks. TABLE IV summarizes the combination of locks and dataset where the shared classifier will be used.

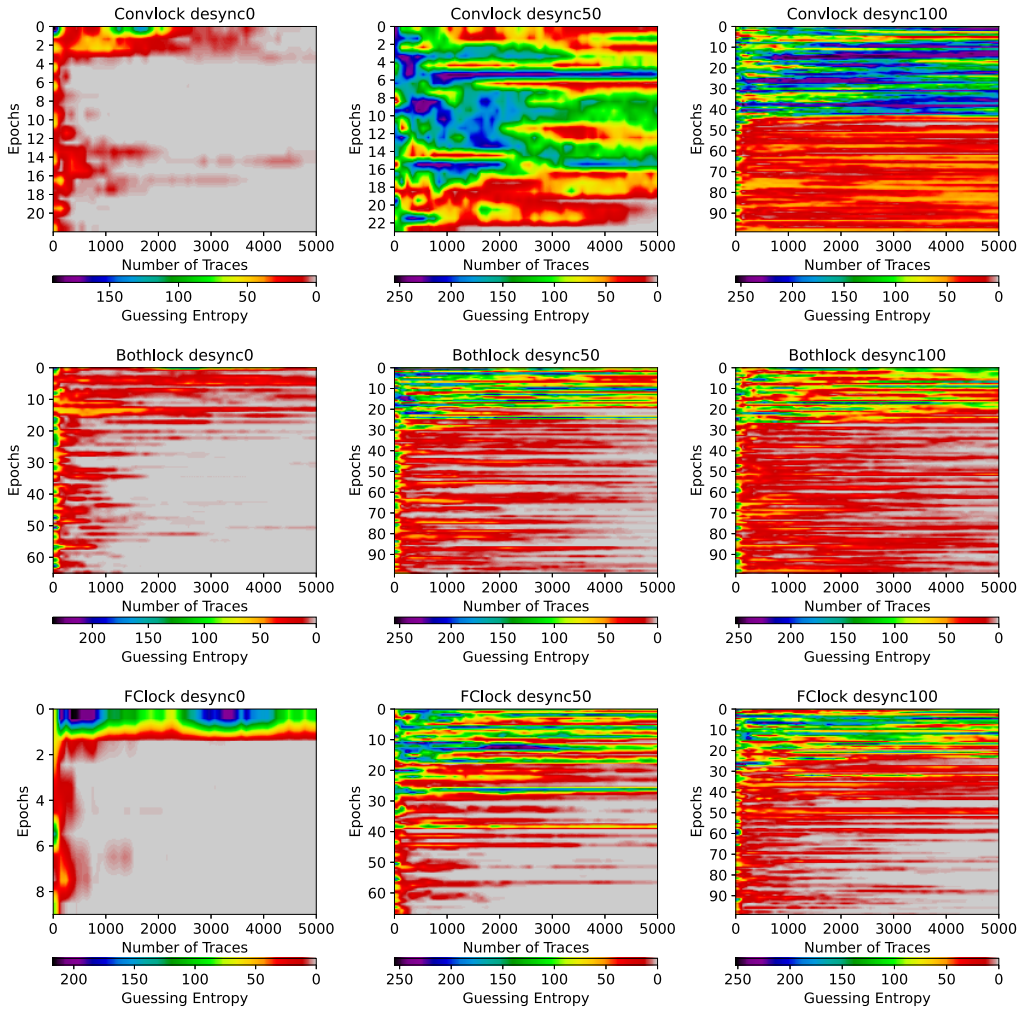

Fig. 11. The training results of the knowledge transferability experiments. Through the columns lies the levels of desynctronization [0, 50, 100]; while through the rows, lies the different block lock cases —ConvLock, BothLock, and FCLock.

We previously said that the chosen classifier ASCAD^f *desync*50 will tackle a more complex dataset —the ASCAD^r *desync*100. Now, by evaluating the ASCAD^r *desync*0 dataset; then, we will cover the scenario where the shared classifier comes from a more complex dataset. Still, bear in mind that it is in terms of desynchronization because it does not come from a complex dataset in terms of its secret key's nature —from random to fixed key, for example. So, we rate the "experience" of the classifier as *medium level of experience*.

Due to space constraints, we did not perform an inter-classifier sharing and a no-block lock sharing protocol; furthermore, we claim that the sharings addressed in our experiments represent the difficult one, being enough to prove our contribution. However, we let those experiments and further combinations of sharing protocol for future works. Fig. 11 depicts the training process of all chosen sharing protocols. It is worthy of mentioning that we did not change the loss intensity parameter (γ), reducing the effort in tuning the modular network.

For this experiments, we trained the modular networks using the early stopping framework from [24]. Contrary we did in the previous section, we do stop the training when the policy finds out the best learning state. We can now know the number of epochs required to achieve good performance.

Generally, all sharing protocols perform well if we contrast the training process of Fig. 11 and Fig. 5. Nevertheless, the sharing protocols that worked best are the *fully-connected block*, *both blocks*, and the *convolutional block lock*.

Observe that for the fully-connected block lock, ASCAD^r *desync*0 has a convergent guessing entropy after 9 epochs, ASCAD^r *desync*50 at 65 epochs, and ASCAD^r *desync*100 took the whole training process (100 epochs); even thought, it achieves good performance. Both blocks lock cases seem to require more epoch or the convergence is roughly achieved, ASCAD^r *desync*0 and ASCAD^r *desync*50, for instance. Finally, we notice that the convolutional block lock converges after several more epochs than the previous locks. In this case, ASCAD^r *desync*100 did not converge within 1 000 leakage traces. We summarize in Fig. 12 the best guessing entropy from all combination of locks.

C. Discussion

Using a shared classifier instead of a non-trained modular network, we have reduced the training time and the effort in tuning hyperparameters while evaluating the leakage of

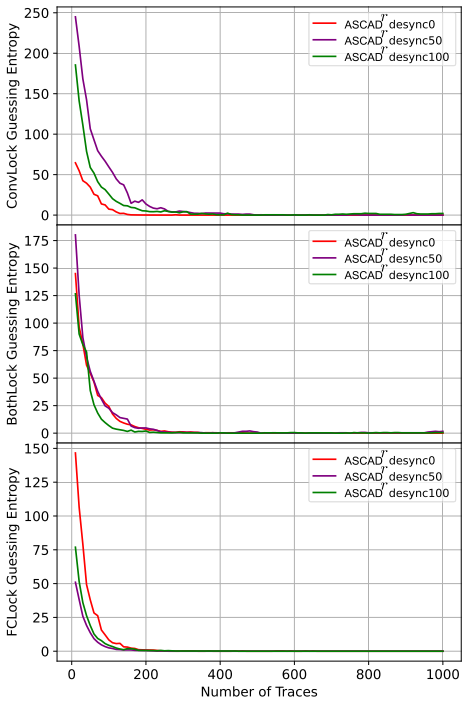

Fig. 12. Best guessing entropy results from all sharing protocols.

a dataset with good results. Since we locked some blocks and the whole classifier, we reduced the number of neurons to train; consequently, the training time is reduced since the number of operations per neuron is less than a non-trained modular network. As we do not have to tune the hyperparameter of a classifier, then we do not spend time in it. Further, we are confident that the classifier has a high probability of working since it already has previous "experience". We demonstrated the latter by actually achieving good results.

Clearly, some initial effort has to be made, for instance, to tune the latent space and losses intensity hyperparameters. Coming up with an initial deep learning modular network could be challenging, but it is an equivalent effort in finding several small deep learning models for different datasets. Finally, when saying that a classifier has previous experience, we do not claim that it will work flawlessly. As we said, the experience of a shared classifier represents a neurons' weights initializer. So instead of randomly initializing the weights using any well-known function —he uniform, for instance. We start from a state leveraged by a previous worked learning. We have demonstrated, experimentally, that it has good results.

VIII. Conclusion

We introduced the DL-SCA modular network approach to conducting SCA evaluation reusing modules from previously trained modular networks. A DL-SCA modular network consists of two main modules; a downsampler and a classifier. We demonstrate that modules from a modular network can be detached and attached to other modular networks and conduct an efficient SCA evaluation. The strategy is to use a classifier with good performance and reuse it to conduct another evaluation in a different dataset.

Our experiments demonstrate that it is not mandatory to re-train a classifier module to effectively evaluate the aimed dataset, regardless of whether the source classifier has been trained with a dataset with a lower noise level. We systematically lock the layers of the classifier to restrict them from getting trained, replicating different sharing protocols to evaluate the effectiveness of our approach.

As future work, we aim to work with more sharing protocols and improve the performance of our modular network in future works by using other types of deep learning architecture for the downsampler. Furthermore, we look for applying methodologies that might help tune the hyperparameters of a modular network. We plan to perform experiments using more datasets. Also, experiments using more combinations of shared classifiers. For instance, a shared classifier trained in a more complex dataset than the target dataset.

References

[1] S. Picek, A. Heuser, A. Jovic, L. Batina, and A. Legay, "The secrets of profiling for side-channel analysis: Feature selection matters," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 1110, Jan. 2017.

[2] S. Paguada, L. Batina, and I. Armendariz, "Toward practical autoencoder-based side-channel analysis evaluations," *Comput. Netw.*, vol. 196, Sep. 2021, Art. no. 108230.

[3] L. Wu and S. Picek, "Remove some noise: On pre-processing of side-channel measurements with autoencoders," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 4, pp. 389–415, Aug. 2020.

[4] M. Naila, L. Papachristodoulou, A. P. Fournaris, L. Batina, and Y. Kong, "Machine-learning assisted side-channel attacks on RNS ECC implementations using hybrid feature engineering," in *Proc. 13th Int. Workshop Constructive Side-Channel Anal. Secure Design (COSADE)*, in Lecture Notes in Computer Science, Leuven, Belgium, vol. 13211, J. Balasch and C. O'Flynn, Eds. Springer, Apr. 2022, pp. 3–28, doi: 10.1007/978-3-030-99766-3_1.

[5] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, vol. 31. Cham, Switzerland: Springer, 2008.

[6] N. Mukhtar and Y. Kong, "On features suitable for power analysis—Filtering the contributing features for symmetric key recovery," in *Proc. 6th Int. Symp. Digit. Forensic Secur. (ISDFS)*, Mar. 2018, pp. 1–6.

[7] N. Mukhtar, A. P. Fournaris, T. M. Khan, C. Dimopoulos, and Y. Kong, "Improved hybrid approach for side-channel analysis using efficient convolutional neural network and dimensionality reduction," *IEEE Access*, vol. 8, pp. 184298–184311, 2020.

[8] M. O. Choudary and M. G. Kuhn, "Efficient, portable template attacks," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 2, pp. 490–501, Feb. 2018.

[9] L. Lerman, R. Poussier, O. Markowitch, and F.-X. Standaert, "Template attacks versus machine learning revisited and the curse of dimensionality in side-channel analysis: Extended version," *J. Cryptograph. Eng.*, vol. 8, no. 4, pp. 301–313, Apr. 2017.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[11] S. Paguada and I. Armendariz, "The forgotten hyperparameter: Introducing dilated convolution for boosting CNN-based side-channel attacks," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.* Cham, Switzerland: Springer, 2020, pp. 217–236.

[12] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Advances in Cryptology EUROCRYPT 2009*, A. Joux, Ed. Berlin, Germany: Springer, 2009, pp. 443–461.

[13] G. H. Dunteman, *Principal Components Analysis*, no. 69. Newbury Park, CA, USA: Sage, 1989.

[14] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.

[15] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, "Methodology for efficient CNN architectures in profiling attacks," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 1–36, Nov. 2019.

[16] E. Cagli, C. Dumas, and E. Prouff, "Kernel discriminant analysis for information extraction in the presence of masking," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.* Cham, Switzerland: Springer, 2016, pp. 1–22.
[17] E. Cagli, C. Dumas, and E. Prouff, "Enhancing dimensionality reduction methods for side-channel attacks," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.* Cham, Switzerland: Springer, 2015, pp. 15–33.
[18] E. Cagli, "Feature extraction for side-channel attacks," Ph.D. dissertation, Sorbonne Univ., Informatique, Télécommunications, Électronique De Paris, Paris, France, 2018.
[19] S. Picek, A. Heuser, A. Jovic, and L. Batina, "A systematic evaluation of profiling through focused feature selection," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 2, pp. 2802–2815, Sep. 2019.
[20] G. Yang, H. Li, J. Ming, and Y. Zhou, "CDAE: Towards empowering denoising in sidechannel analysis," in *Proc. Int. Conf. Inf. Commun. Secur.*, 2020, pp. 269–286.
[21] E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Canovas, "Study of deep learning techniques for side-channel analysis and introduction to ASCAD database," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 53, Jan. 2018.
[22] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin, Germany: Springer-Verlag, 2002.
[23] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of AES," in *Selected Areas in Cryptography*. Berlin, Germany: Springer, 2004, pp. 69–83.
[24] S. Paguada, L. Batina, I. Buhan, and I. Armendariz, "Being patient and persistent: Optimizing an early stopping strategy for deep learning in profiled attacks," 2021, *arXiv:2111.14416*.
[25] D. Zhang, H. Wang, M. Figueiredo, and L. Balzano, "Learning to share: Simultaneous parameter tying and sparsification in deep learning," in *Proc. Int. Conf. Learn. Represent.*, 2018. [Online]. Available: https://openreview.net/forum?id=rypT3fb0b
[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[27] L. Masure, C. Dumas, and E. Prouff, "Gradient visualization for general characterization in profiling attacks," in *Proc. Int. Workshop Constructive Side-Channel Anal. Secure Design*. Cham, Switzerland: Springer, 2019, pp. 145–167.
[28] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: Learning important features through propagating activation differences," 2016, *arXiv:1605.01713*.
[29] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross, "Towards better understanding of gradient-based attribution methods for deep neural networks," 2017, *arXiv:1711.06104*.
[30] M. Müller, "Dynamic time warping," in *Information Retrieval for Music and Motion*. Springer, 2007, pp. 69–84, doi: 10.1007/978-3-540-74048-3_4.

Servio Paguada received the B.Eng. degree in system engineering and the B.Sc. degree in mathematics with informatics applications from the Universidad Nacional Autonoma de Honduras in 2011, the master's degree in information technology management from Universidad Tecnologica Centroamericana in 2014, and the M.Sc. degree in embedded systems from Mondragon University, Basque, Spain, in 2016. He is currently pursuing the Ph.D. degree with Radboud University, Nijmegen, The Netherlands. He is also part of the Ph.D. Students Group, IKERLAN Technology Research Centre, Arrasate-Mondragón, Gipuzkoa, Spain. His current research interest includes optimizing profiled side-channel analysis applied to embedded systems.

Lejla Batina (Senior Member, IEEE) received the Ph.D. degree from KU Leuven, Belgium, in 2005. She has studied a Professional Doctorate in engineering at the Eindhoven University of Technology in 2001. From 2001 to 2003, she worked as a Cryptographer at SafeNet B.V. She is currently a Professor in embedded systems security at Radboud University, Nijmegen, The Netherlands. Her research group consists of 12 researchers and eight Ph.D. students have graduated under her supervision. She is an Editorial Board Member of top journals in *Security*, IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, and *ACM Transactions on Embedded Computing Systems*.

Ileana Buhan received the Ph.D. from Twente University, The Netherlands, in 2008. She is currently an Assistant Professor of cryptographic engineering at the Digital Security Group, Radboud University. She spent over ten years in the security evaluation industry. Her research interest focuses on developing tools to help designers of cryptographic algorithms develop secure implementations. She serves on several program committees of conferences that specialized on hardware security, such as TCHES, COSADE, CARDIS, FDTC, DATE, and SPACE. She was also appointed as the General Chair of CHES 2018 and the Program Co-Chair for CARDIS 2022.

Igor Armendariz received the Ph.D. degree from the University of the Basque Country in 1996. He had a Scholarship Award by CEIT from 1991 to 1994, as a Researcher at the IKERLAN Technological Research Center, from 1995 to 1996. He was a Professor in computer science at the University of the Basque Country from 1996 to 2000. He began working at the Communication Department, IKERLAN Research Center, from 2000 to 2015. He is currently a Researcher at the IKERLAN Technological Research Center within the Cybersecurity in Embedded Systems Team. He is also part of the Industrial Cybersecurity Department, IKERLAN Research Center. He is a Cybersecurity Specialist Acc. to IEC-62443-4-1 and IEC-62443-4-2 (TÜV Rheinland, Components) #230/19. He is co-supervising a couple of Ph.D. students working with side-channel attacks and countermeasures.