A Practical Tutorial on Deep Learning-based Side-channel Analysis

Sengim Karayalçin, Marina Krček, Stjepan Picek, *Senior Member, IEEE*

***Abstract*—This tutorial provides a practical introduction to Deep Learning-based Side-Channel Analysis (DLSCA), a powerful approach for evaluating the security of cryptographic implementations. Leveraging publicly available datasets and a Google Colab service, we guide readers through the fundamental steps of DLSCA, offering clear explanations and code snippets. We focus on the core DLSCA framework, providing references for more advanced techniques, and address the growing interest in this field driven by emerging standardization efforts like AIS 46. This tutorial is designed to be accessible to researchers, students, and practitioners seeking to learn practical DLSCA techniques and improve the security of cryptographic systems.**

***Index Terms*—Deep Learning, Side-channel Analysis, Evaluation**

I. Introduction

Cryptographic algorithms are essential in data communication systems across various applications, ensuring the secrecy of sensitive and classified information. Cryptography employs mathematical principles to enable data confidentiality, integrity, and authenticity (among other goals), with numerous algorithms designed to be mathematically secure [1], [2]. However, these algorithms must be implemented on a certain software or hardware platform. This implementation process introduces specific characteristics and vulnerabilities. While classical cryptoanalysis focuses on identifying theoretical weaknesses within cryptographic algorithms, physical attacks focus on the implementation of the algorithm [3], [4], [5]. Implementation attacks aim to retrieve secret information or bypass security measures by exploiting vulnerabilities specific to how cryptographic algorithms are implemented in practice. Such attacks are powerful and represent a critical concern for cryptographic system developers. Consequently, these threats are explicitly addressed within the Common Criteria (CC) [6] for evaluating the security of cryptographic systems.

Implementation attacks can be categorized based on their level of invasiveness on the device, ranging from invasive, semi-invasive, to non-invasive, and their impact on the device's operation, classified as either active or passive attacks. This tutorial focuses on passive and typically non-invasive attacks, known as side-channel analysis (SCA). Moreover, we concentrate on attacks on cryptographic algorithms, where the goal is to extract the secret information (key).

S. Karayalçin is with Leiden University, the Netherlands
M. Krček and S. Picek are with Radboud University, Nijmegen, The Netherlands.
S. Picek is the corresponding author (stjepan.picek@ru.nl)

Side-channel analysis exploits unintended side-channel leakage that occurs during the execution of cryptographic algorithms on targeted devices. This side-channel information includes power consumption [7], electromagnetic emissions [8], execution time [3], or even acoustic signals [9]. By analyzing these side channels, attackers can potentially extract sensitive information such as cryptographic keys.

Depending on the attacker's ability and resources, SCA can be divided into non-profiling (direct) and profiling (two-stage) SCA. Non-profiling SCA analyzes traces (depending on the specific attack, it can be from one trace to millions of traces) from a target device directly, relying on statistical methods (e.g., Pearson correlation, difference-of-means, or mutual information) to reveal correlations between observed side-channel information and secret data. Examples include techniques such as Simple Power Analysis (SPA) [3] and Differential Power Analysis (DPA) [7]. On the other hand, a two-stage or profiling SCA assumes a more powerful adversary with access to an open device identical (or at least similar) to the target, allowing a detailed profiling model to be created. These attacks are performed in two phases: the first phase builds a model using the clone device under control, and the second phase utilizes the model to obtain the secret from the actual target device. One such attack called template attack (TA) [5] creates a statistical model or 'template' that describes the side-channel leakage and noise of the open (clone) device under control. During the attack phase, the adversary or security analyst uses the created template to analyze the measurements collected from the target device, which should be (at least) similar to the clone device, and determines the most likely key used on the target device.

Template attacks, known for their effectiveness from an information theoretical perspective [5], require significant feature engineering and numerous traces to model device leakage accurately. As such, researchers are increasingly looking toward advanced methods such as Machine Learning (ML) to enhance side-channel analysis. Initial explorations utilized traditional machine learning techniques such as Support Vector Machines (SVM) [10], [11] and Random Forest [12], followed by the application of Multilayer Perceptron (MLP) [13], [14]. The introduction of deep learning-based profiling SCA marked a significant development, employing various architectures like MLP, Stacked Autoencoder, Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN) to analyze side-channel data [15]. Since 2016 [15], deep learning methods have been extensively studied for side-channel analysis applications and enhancements [16]. These efforts culminated when the German Federal Office for Information Security

(BSI) published an updated version of the Application Notes and Interpretation of the Scheme (AIS) 46 in February 2024, which provides guidelines for evaluating machine learning-based side-channel attack resistance [17]. Given these recent developments, this tutorial provides a timely and relevant introduction to the execution of deep learning-based side-channel analysis (DLSCA), adhering to the AIS 46 guidelines. More precisely, we analyze the literature for the period 2016-2024 for the best practices in DLSCA.

To our knowledge, there are no published tutorials on deep learning-based SCA. On the other hand, we note several surveys [18], [19], [20], [21], [22], [23], one systematization of knowledge [16], and a number of publicly available tools for DLSCA [24], [25], [26]. Moreover, the authors in [27] provide a tutorial on several physical attacks, including classical side-channel analysis.

A. Motivation

Up to now, there are a few hundred papers dealing with DLSCA [16]. Most of those works aim to improve the attack performance by concentrating on more powerful AI approaches, for instance, by improving hyperparameter tuning [28] or data augmentation [29]. Some, on the other hand, go in the direction of considering different threat models [30], [31] or developing new types of attacks [32]. While such developments push state-of-the-art forward, they also make it more difficult for novices in the SCA domain to start their research. Indeed, to be an independent DLSCA researcher, one needs to have good knowledge of both side-channel analysis and deep learning. Moreover, proficiency in Python and some deep learning frameworks like `PyTorch`, `TensorFlow`, or `Keras` is necessary. As such, it is difficult to expect such knowledge even from more experienced researchers, let alone beginners. This tutorial aims to provide all foundational (practical) information for beginners to be able to conduct deep learning-based side-channel analysis.

For the tutorial, we use Google Colab [33]. Colab is a hosted Jupyter Notebook [34] service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Therefore, it should be possible to run the notebook as provided. The tutorial notebook can also be downloaded as a Jupyter Notebook and used in a local Python environment. Still, due to the ease of use, we implement our DLSCA framework in Colab and provide it in the GitHub repository.[1] Note that the code snippets within this work might rely on some existing code in the Colab notebook, and thus copy-pasting every code snippet into a new Colab or local Jupyter notebook will not be fully executable unless the additional code from the Colab notebook itself is also used.

B. Prior Knowledge

We do not assume that the reader has previous familiarity with side-channel analysis or deep learning. We briefly explain all the relevant concepts in this work. More precisely, we cover the main concepts while leaving more advanced methods and techniques for interested readers to explore.

For the practical part of the tutorial, we assume the reader has a basic understanding of Python. This includes the ability to read and write Python code, including fundamental constructs such as loops, functions, and classes. Familiarity with `Numpy` library is also expected. While prior experience with deep learning frameworks (such as `Keras`, `TensorFlow`, or `PyTorch`) is helpful, it is not strictly required. Our explanations will clarify the functionality of the `TensorFlow` library used in this tutorial, including tasks such as training a model and performing inference, ensuring that readers without prior deep learning experience can comprehend the tutorial. However, for details, we refer the reader to the official library documentation.[2].

C. Information on Google Colab Service

We provide an executable tutorial using Google Colab, a free online platform that requires a Google account (easily created at no cost). Google Colab allows users to run code using CPU, GPU, or TPU units. The platform includes a pre-configured Python environment with many commonly used libraries already installed. If any required libraries are missing, they can be installed within the notebook. However, note that while the libraries are immediately available without installing them, they must be explicitly imported into the Python code to be used. The Colab notebook contains three types of cells: code cells, text cells, and header cells (also a text cell). To insert new text or code cells, use the options provided in the **Insert** tab. To execute a cell, click the "Run" button on the top left side of the cell or press Ctrl+Enter when the cell is selected. Additional execution options are available in the notebook's header under the **Runtime** tab. To utilize GPU or TPU units for training and running deep learning models, check the option *Change runtime type*. Note that, with the free account, there are time and memory limits to using these computational units.

D. Outline

This paper is organized as follows. Section II provides a relevant introduction to side-channel analysis and deep learning, emphasizing deep neural networks. Next, Section III provides information about deep learning-based SCA, common threat models, and a visual flowchart of the DLSCA analysis. Each part of the flowchart is discussed and explained in a separate section. As such, this paper can be studied fully in the order given, or readers can go into separate sections, depending on their goals. Section IV discusses relevant preprocessing steps, such as normalization and standardization. Section IV-A elaborates on common feature engineering options. Section V discusses commonly used neural network types, data augmentation, and hyperparameter tuning options. Section VI presents options when mounting the attack phase and results evaluation. Section VII discusses common options for AI interpretability/-explainability in the context of DLSCA. Finally, Section VIII

[1]https://github.com/marinakrcek/DLSCA-tutorial

[2]https://www.tensorflow.org/api_docs

concludes the paper. We note that every section is accompanied by relevant source code snippets, and we explain the basic approaches used in DLSCA. Moreover, we briefly list more advanced options for interested readers to explore.

II. Background

A. Side-channel Analysis

SCA poses a major threat to devices handling sensitive data like keys, private certificates, and intellectual property. More precisely, and in the context of cryptographic implementations, in (physical) SCA, we attempt to extract secret information from side-channel traces, e.g., power/electromagnetic (EM) measurements [7], [8], during the computation of a cryptographic algorithm. There, for n encryptions[3] we collect n traces of m samples (features/points of interest) resulting in traces $\mathbf{X} = \{x^j, 1 \leq j \leq n\}$ where x_j is a vector with m points. Then, for each of these traces, key(s) k^j and plaintexts p^j allow us to generate a set of measurement labels.

In the rest of this work, we will consider the NIST Advanced Encryption Standard (AES) cipher [35], which is the algorithm of choice for most settings when encrypting information and is also the common target to explore in the SCA literature [16]. AES is a byte-oriented cipher that operates in a number of rounds and where each of the rounds contains several operations. A common place to attack is after the `S-box` part, making the function of interest $IV = \texttt{S-box}(plaintext \oplus key)$. We can use the divide-and-conquer approach and consider attacking every key byte separately (as AES is byte-oriented); we denote the i-th byte of the key and plaintext as k_i^j and p_i^j, respectively. The intermediate value then equals $IV^j = \texttt{S-box}[p_i^j \oplus k_i^j]$. Finally, when modeling leakage, it is common to assume a certain behavior of how the device leaks, a concept known as the leakage model. Common leakage models include the Hamming weight (HW) leakage model[4], which assumes the leakage is proportional to the number of ones in a byte, the least/most significant bit (LSB/MSB) that assumes the leakage happens in a single bit only, and the identity (ID) model that assumes that the leakage is proportional to the value at the output of the `S-box`. Note that since AES is byte-oriented, the `S-box` output contains 256 values, and the Hamming weight (distance) of those values can be between 0 and 8 (making a total of 9 values). While the ID leakage model is bijective, the HW/HD leakage models follow a binomial distribution. To assess the attack's effectiveness, it is common to consider how many guesses one needs to make before finding the correct key. As such, the fewer guesses, the better the attack [16]. Thus, the simplest metric, key rank (KR), states in what position in the key guessing vector (a vector of all possible keys sorted from the most likely to the least likely guess) is the correct key. A somewhat more complex metric is the guessing entropy (GE) [36], which is the average key rank. The averaging is done to improve the statistical quality (i.e., to reduce the effect of specific traces used) of the attack. Finally, the success rate (SR) of order o is a metric indicating whether in the first o guesses is the correct key.

Some metrics aim to quantify how much information is extracted by neural networks. Perceived Information [37] is a lower bound for mutual information which can be easily computed from model predictions. Several works have explored theoretical properties [18] and proposed enhancements [38], [39]. Note that the goal of an SCA attack is to minimize the key rank and guessing entropy values and maximize the success rate and perceived information metric.

Recall that a common SCA division is into direct and profiling attacks [16]. Direct attacks assume a single device where the attacker uses statistical techniques (called distinguishers) to find the most likely keys. A common approach is the Correlation Power Analysis (CPA) [40].

In profiled attacks, one assumes the attacker can access a copy of a device to be attacked. This copy is under the complete control of the attacker and is used to build a model of the device. Then, the attacker uses that model to attack a different (but similar) device. While the profiled attack is more complex due to the assumption of access to a copy of a device, it can be significantly more powerful than direct attacks. Indeed, provided that the model is well-built, one could need as little as a single trace from the device under attack to obtain the secret key. Direct attacks may need millions of traces to break a real-world target [16]. One can easily observe a similarity between profiled attacks and the supervised machine learning paradigm (where building a model is training, and the attack is testing). Consequently, in the last decade and more, many machine (deep) learning algorithms have been tested in SCA.

One commonly uses countermeasures to protect against SCA, which can be either hiding or masking. In both cases, the goal is to remove the correlation between the observed quality (traces) and secret information. Hiding countermeasures can happen in the amplitude domain by randomizing/smoothing the signal or by adding desynchronization/random delays in the time domain. Masking [41], on the other hand, divides a secret variable into a number of shares such that to obtain the secret information, an attacker needs to combine all the shares. For further information about SCA, refer to [42].

B. Deep Learning

Deep learning, a subfield of machine learning, is characterized by its ability to construct models that learn increasingly abstract representations of data compared to traditional machine learning approaches [43]. This capability to perform advanced feature extraction and transformation allows for more effective high-dimensional data processing to discover complex patterns. Traditional machine learning models, such as linear regression, logistic regression, naive Bayes, and decision trees, are designed to learn from data in a straightforward manner, often requiring manual feature selection and data preprocessing [44], [45]. Artificial Neural Networks (ANNs) originate from the domain of traditional machine learning, beginning with the fundamental concept of the Perceptron.

[3]For simplicity, we mention only encryption; the process is analogous for decryption.

[4]Or the Hamming Distance (HD) leakage model that assumes the leakage is proportional to the number of transitions from zero to one and one to zero.

This simple model, representing a single neuron, enabled the development of more complex structures such as the multilayer perceptron, which consists of multiple interconnected neurons organized in layers. There is no consensus about how much depth (layers and neurons) a model requires to be considered a deep learning model. However, deep learning extends ANNs to Deep Neural Networks (DNNs), which include different architectures, such as the mentioned MLP, but also the convolutional neural network, the recurrent neural network and more. These architectures enable automatic extraction and learning of complex patterns in data.

Depending on what kind of setting is assumed during the learning process of the machine learning algorithms, we can divide them into supervised learning, semi-supervised learning, reinforcement learning, and unsupervised learning. Unsupervised learning commonly has access to an unlabeled dataset and aims to learn useful properties of the structure or relationships within the data. On the other hand, supervised learning has a dataset with the associated labels and aims to learn a mapping function that accurately predicts the output for new, unseen data. This setting is common for classification and regression tasks. Classification is a type of task where the algorithm is required to specify to which category (class) some input belongs, while in regression, the algorithm predicts a numerical value given some input. Semi-supervised learning assumes only a small number of measurements with associated labels, making this setting come between supervised and unsupervised learning. Reinforcement learning consists of intelligent agents that interact with an environment and learn through feedback in the form of rewards or penalties associated with taken actions. In the context of deep learning-based SCA, it is common to use a supervised learning paradigm with the classification task, and as such, we will concentrate on it in the rest of this work.

1) Supervised learning: To understand DLSCA, we explain the supervised learning process with neural networks. Given a labeled training set of inputs x and labels y, a supervised learning algorithm aims to learn a function f that maps from x to corresponding y. The goal of the learning algorithm is that the final model can generalize from the training data to unseen data in predicting output y. In classification tasks, the output y is a category given to the input x. Classification tasks can be binary, i.e., two possible categories, or multi-class with an arbitrary number of categories.

Given a set of N training examples of form $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ such that x_i is the feature vector of the i-th example and y_i its corresponding label (class), a learning algorithm tries to learn a function $f : X \rightarrow Y$, with X as the input space, and Y as the output space. To measure how well the function fits the training data, a loss function L is defined. The loss of predicting value $\hat{y}$ for input example (x_i, y_i) is $L(y_i, \hat{y})$. The model is then trained through a learning process to minimize the error (loss function) by adjusting its parameters (commonly denoted by θ) using optimization methods like gradient descent [46], [43]. The model is tested on unseen labeled data (test data) to measure how well it generalizes using metrics like accuracy (for classification) or mean squared error (for regression). Lastly, with a well-working model on test data, we can use the trained model to make predictions on new, unseen data without the labels.

2) Neural Networks: Neural networks consist of an input layer, one or more hidden layers, and an output layer. The input layer receives data, which is a set of numbers representing images or signals. Hidden layers process the data to extract patterns and features and create a hierarchy of abstractions. The output layer provides the result, which in regression will be a specific value, and, in classification, will be (most often) a probability distribution over the possible classes. Layers are made up of neurons, which are the basic computational units of the network. During the learning process, the network takes in the numerical representation of the input data; in SCA, this would be the traces, which are a list of numbers representing, e.g., the power measurements during the encryption/decryption. The input data goes through the neurons, where each neuron applies a weight to the input, and these weights determine how important each input is. The bias term enables a neuron to activate even when the weighted sum of inputs is zero, providing crucial flexibility in the model's ability to learn complex patterns. We can represent the neuron operation with Eq. (1):

$$z = (w_1 \times x_1) + (w_2 \times x_2) + \ldots + b, \tag{1}$$

where $w_1, w_2, \ldots$ are weights, b is bias, and $x_1, x_2, \ldots$ are inputs. Note that weights and bias are trainable parameters of the network. As one can notice, neuron computation is a linear transformation, and therefore, the learning process is limited to linear functions. To resolve this, nonlinearity can be added with activation functions. The output of the neuron z goes through an activation function where common functions are ReLU (defined as $ReLU(z) = max(0, z)$) and sigmoid (defined as $\sigma(z) = \frac{1}{1+e^{-z}}$). The output of the activation function from one layer becomes the input for the following layer. Lastly, the output layer for classification tasks has one neuron per class, and the output values are converted into probabilities using a function like $softmax$ (defined as $softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$, where K is the number of classes x_i is the output of the i'th neuron). Then, the prediction of the network would be the class with the highest probability. With regression, the prediction of the network would be the output of the single neuron of the last output layer. For the learning process to occur, we train the network to minimize the difference between the predicted output and the true values (labels) present in our data in a supervised setting. The difference is measured using the loss function. For classification tasks, that is usually cross-entropy. Note that this process, including the loss calculation, is called the forward pass of the network, as the input data is processed from the input layer to the output layer.

Once the loss function is calculated, the network learns by adjusting the trainable parameters (weights and biases). Backpropagation is an efficient algorithm used to train most artificial neural networks. It consists of the previously mentioned forward pass and a backward pass. The backward pass consists of computing the gradient of the loss with respect to a trainable parameter in the neural network. This is done by taking partial

derivatives for trainable parameters in the output layer, and then iteratively using the chain rule to compute gradients for parameters in earlier layers. Using this gradient, we can then update the weights and biases to lower the error the network is making in its predictions. Gradient descent is used as an optimization algorithm that minimizes the loss by adjusting the trainable parameters in the direction of decreasing error. There are several optimization methods that can be used, a popular choice is the Adam optimizer [47]. Training of a neural network is done for a number of epochs, i.e., full iterations over all the samples in the training dataset. As computing gradients for the full dataset is challenging computationally, training is done by repeatedly sampling small subsets of the dataset, which we refer to as batches.

Code sample. Here, we consider a simple multilayer perceptron, which is a neural network in which all the neurons in each layer are connected to all the neurons in the following layer throughout the whole network. Listing 1 shows how the architecture of an MLP model can be created using the `TensorFlow` library. `Sequential` class groups a stack of layers into a model. In this example, we arrange a list of our layers in the order we want the model to be. To use the fully connected layers, we use the `Dense` class. Thus, in our example, we have an input with 100 features that are passed to the first hidden layer with 64 neurons, while the second hidden layer has 32 neurons. Both hidden layers are defined with the `ReLU` activation function. The output layer has three neurons, corresponding to the three possible classes in this example. As mentioned before, we use `softmax` to transform the output values of the neurons in the last layer to a probability distribution.

Listing 1. Creating the architecture of a simple MLP.
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

input_dim = 100
model = Sequential([
    # hidden layers
    Dense(64, activation='relu', input_shape=(
        input_dim,)),
    Dense(32, activation='relu'),
    #output layer
    Dense(3, activation='softmax')
])
```

Once the architecture is created, we call the `compile` function on the model to configure the model for training. We need to define other hyperparameters for the training, such as the optimizer, learning rate, and loss function. Listing 2 shows setting the popular `Adam` optimizer with learning rate $1e^{-3}$, categorical cross-entropy as loss function, and the last argument sets a list of metrics to be evaluated by the model during training and testing, which in this example is only the accuracy of the model's predictions.

Listing 2. Compiling the model.
```
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(learning_rate=1e-3),
    loss='categorical_crossentropy', metrics=['
    accuracy'])
```

Finally, we can train the model with the code shown in Listing 3. The whole training process using explained back-propagation is abstracted in the function `fit` on the model class. The `fit` function takes in the data X_train with the corresponding labels y_train. We set the number of epochs to 10 and the batch size to 32, meaning that the training will iterate the whole training set 10 times in batches of 32 samples.

Listing 3. Training the model.
```
model.fit(X_train, y_train, epochs=10, batch_size
    =32)
```

The trained model is evaluated on data not used in the training process with a function `predict` as shown in Listing 4. The predictions will hold an array of probability values for each label for each sample in X_test. Thus, to obtain the predicted label, we take the index of the class with the highest predicted probability.

Listing 4. Model preditions (inference).
```
predictions = model.predict(X_test)
predicted_classes = predictions.argmax(axis=1)
```

The function `evaluate` can return the loss and accuracy metrics on the given dataset (X_test, y_test) as shown in Listing 5.

Listing 5. Model evalaution.
```
loss, accuracy = model.evaluate(X_test, y_test)
```

For more information on the `TensorFlow` library, we refer the reader to the `TensorFlow` documentation [48].

3) Difference Between Unsupervised and Non-profiled Settings: While it may seem that the unsupervised learning in ML is the same as the non-profiled setting in SCA, there are some differences. An unsupervised setting means the training dataset has no associated labels. On the other hand, in a non-profiled setting, we assume the attacker does not have a priori knowledge about the labels but instead guesses them. In that way, the attacker still uses the supervised learning paradigm. On the other hand, supervised machine learning and profiled SCA follow the same scenario.

III. Deep Learning-based Side-channel Analysis (DLSCA)

This section outlines the main characteristics of the profiled DLSCA framework and provides supporting code snippets to ensure a practical tutorial. Note that non-profiled DLSCA is not considered here, as it represents a less explored (at least at the moment) scenario and a separate tutorial for the non-profiled case would be more appropriate. However, the principles are similar, where models are trained in the same way as for profiling DLSCA, but instead of a labeled training data set with a known correct key, key hypotheses are used to assign labels [49]. This work trains the model 256 times (for each possible 8-bit key hypothesis), while later works propose methods to decrease the expensive computational costs of training multiple neural networks [50], [51], [52], [53].

Figure III represents the DLSCA flowchart with relevant steps, from raw measurements acquisition and profiling phase to the attack evaluation. Each step is described in the following sections.

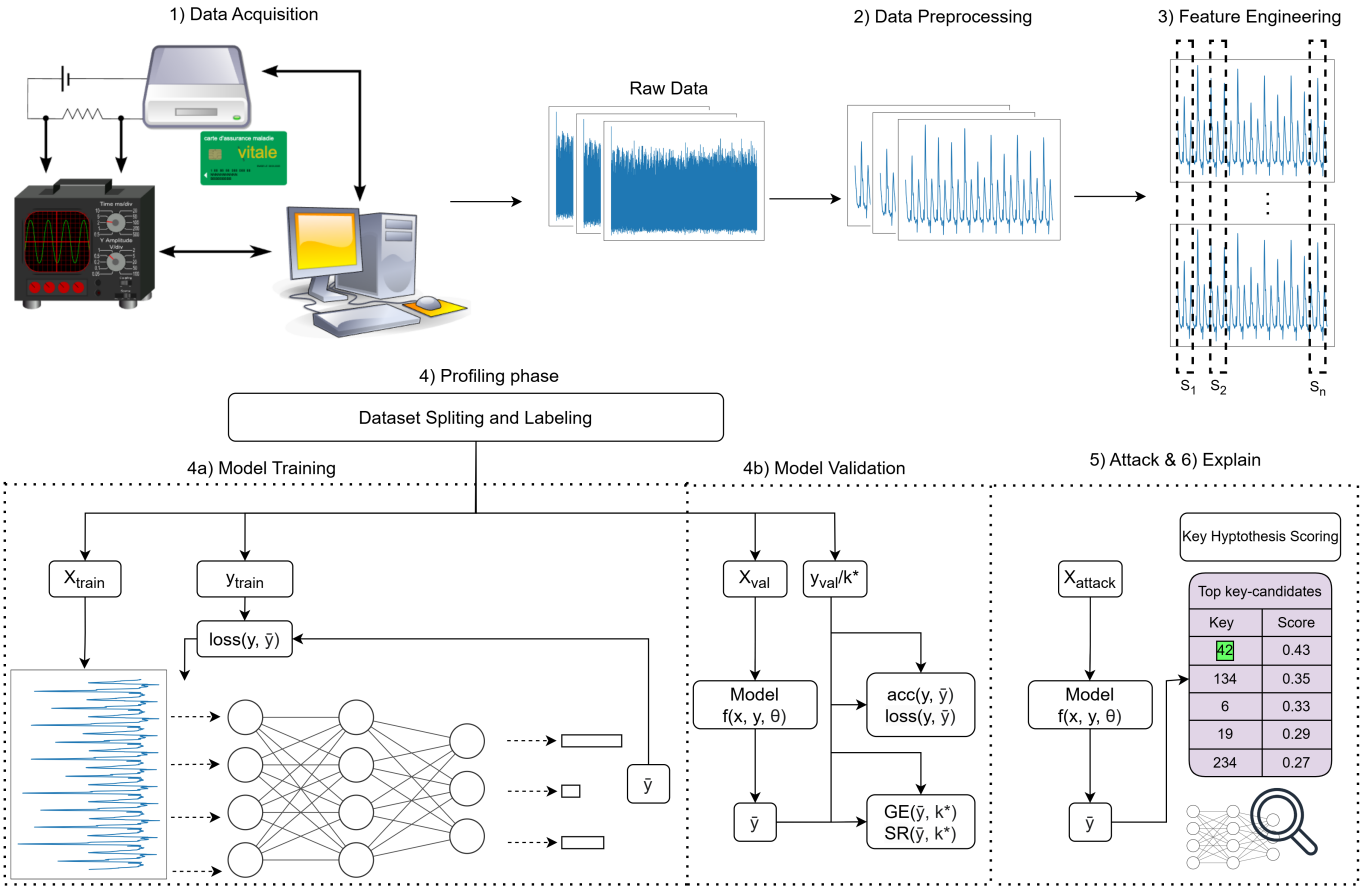

Fig. 1. Deep Learning-based Side-channel Analysis (DLSCA) flowchart.

A. Threat Model(s)

In the DLSCA literature, the profiling step is done on a profiling device. In the classical threat model with the worst-case assumptions [5], an attacker has full control over the profiling device, which allows the attacker to disable countermeasures (e.g., random delays or access to masking randomness). While this setting may be impractical, it simplifies the creation of a profiling model. For masking countermeasures, feature selection becomes relatively straightforward (see the RPOI scenario in Section IV-A), while for hiding countermeasures, it allows for more effective preprocessing to remove the effects of the countermeasures [54].

In general, there are three common threat models in DLSCA. The first, "classical" model assumes that the profiling device and the device under attack are the same. Then, the evaluator measures all measurements from a single device only. Next, the evaluator uses one part of the measurements (commonly consisting of most of the measurements) to build a model (thus, we assume the knowledge of keys, masks, etc.) and the rest of the measurements to simulate the device under attack. Note that this model assumes the worst-case assumptions on the security of the device since it does not account for differences between devices, making the attack simpler. As such, this model is the most commonly used one in both academia and industry [55]. The second model is also known as the "portability" model, which uses different devices for profiling and attack. This model is more difficult but more realistic [56]. Since most of the publicly available datasets do not provide measurements from different devices, not many research works use it. Finally, there is the non-profiled setting [49], where we again work with a single device but do not assume knowledge of keys. This model aims to connect the practicality of direct attacks with the power of deep learning. There are also more advanced threat models used with DLSCA, e.g., weakly profiling [57], scheme-aware [30], blind [58], and collision-based [32].

B. Data Acquisition

As we focus on profiled side-channel analysis, we assume the attacker has access to a clone device of the targeted device. Once the attacker is in possession of a device with full control over it, the first step is to collect the measurements of the side-channel information. Depending on which side-channel information we want to collect, different acquisition equipment might be necessary. Timing attacks exploit the device's running time, power analysis attacks focus on the electric consumption of the device, while electromagnetic (EM) analysis measures the electromagnetic field surrounding the device during its processing.

We depict an acquisition setup in Figure 2, where one can see a target, an EM probe, and an oscilloscope. The topic of data acquisition is complex and constitutes its own

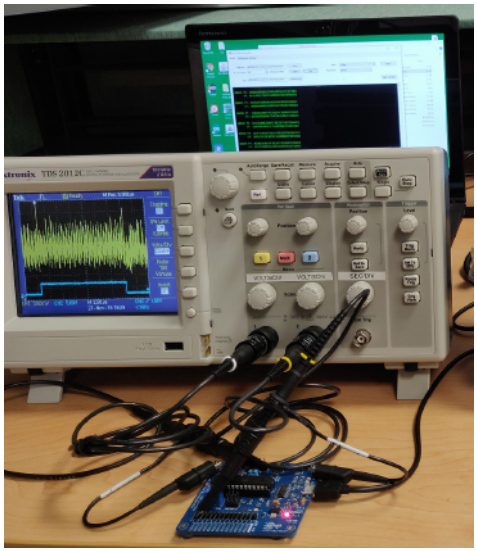

Fig. 2. SCA acquisition setup.

research field. For more information about this topic, we point interested readers to [59], [4], [60]. In the rest of this tutorial, we rely on publicly available datasets, as this is common practice in the DLSCA literature.

Generally, in DLSCA literature, attacks are benchmarked against public targets. This research practice is taken from DL literature where benchmark datasets like CIFAR [61] and ImageNet [62] are convenient tools to drive progress and enable objective assessment of novel methods. The most common target considered in DLSCA literature is the ASCAD dataset [55], and in this tutorial, we will focus on it. While it is currently considered relatively easy to attack with state-of-the-art methods, it is still one of the main targets used in literature. As such, it represents an appropriate target for this tutorial since we do not need to use more complex techniques than the basic DLSCA setup that we aim to showcase. For a relatively recent categorization of public targets, we refer the reader to [16, Section 4.1].

The ASCAD dataset consists of electromagnetic (EM) side-channel measurements from a software implementation of the AES-128 algorithm with Boolean masking protection on ATMEGA8515. The authors [55] provide SCA datasets, with a fixed secret key and variable keys for training. For the attack phase, the traces are correlated with a fixed key in both cases. Moreover, the authors provide raw traces, but also versions of datasets with fewer samples per trace.[5] Raw traces have 100 000 (fixed) or 250 000 (variable) measured samples per trace, while for fixed key version the window of the leaky operation is 700 samples per trace, and for variable key version it is 1400 samples per trace. There are 50 000 profiling (training), 10 000 attack (test) traces for fixed key version, and 200 000 and 100 000 traces for variable key version. We will use the described datasets throughout the tutorial, but the same authors provide desynchronized datasets as well and a newer dataset with affine masking and shuffling instead of Boolean masking as countermeasure [64].

In Listing 6, we show how to download the ASCAD dataset. Note that this will download the dataset in Colab's temporary disk space allocated per session. Colab provides an environment with temporary storage for packages, model checkpoints, and other data that will be deleted once the session is terminated. Each session can last up to 12 hours (for free accounts), after which the resources are recycled and the disk space is cleared. It is not designed to store files long-term.

Listing 6. Downloading ASCAD dataset for tutorial.
```
!wget https://static.data.gouv.fr/resources/ascad-atmega-8515-variable-key/20190903-083349/ascad-variable.h5
```

If we want to avoid downloading the dataset with every session, we can store the dataset in the personal Google Drive service. To use the data stored in Drive, one has to mount it, and Listing 7 shows the code for that. Note also that the data paths might need to be updated.

Listing 7. Mounting the Drive for access to data in your Drive account.
```
from google.colab import drive
drive.mount('/content/drive')
```

Once the data is in our Drive or downloaded directly to the temporary disk space, we can load the datasets. Datasets can be stored in different formats, but a well-accepted one is the Hierarchical Data Formats (HDF) format with file extension `.h5`. The code snippet for loading the ASCAD dataset is shown in Listing 8. Note that this is part of the `load_ascad` function in the tutorial. The function `load_ascad` takes `ascad_database_file` as the argument, which is the path to the ASCAD dataset.

Listing 8. Loading the ASCAD dataset stored in H5 format.
```
check_file_exists(ascad_database_file)
# Open the ASCAD database HDF5 for reading
in_file  = h5py.File(ascad_database_file, "r")
# Load profiling traces
X_profiling = np.array(in_file['Profiling_traces/traces'], dtype=np.int8)
# Load profiling labels
Y_profiling = np.array(in_file['Profiling_traces/labels'])
# Load attacking traces
X_attack = np.array(in_file['Attack_traces/traces'], dtype=np.int8)
# Load attacking labels
Y_attack = np.array(in_file['Attack_traces/labels'])
if load_metadata == False:
    return (X_profiling, Y_profiling), (X_attack, Y_attack)
else:
    return (X_profiling, Y_profiling), (X_attack, Y_attack), (in_file['Profiling_traces/metadata'], in_file['Attack_traces/metadata'])
```

The code snippet shows that the function begins by verifying the dataset file's existence, after which it opens it for reading. The code extracts profiling (file entry 'Profiling_traces/traces') and attack traces (file entry 'Attack_traces/traces'), along with their corresponding labels ('Profiling_traces/labels' and

[5]These are NOPOI and OPOI scenarios, respectively, as discussed in [63], and later in Section IV-A.

'Attack_traces/labels'), which are cryptographic intermediate values. Optionally, it also loads metadata associated with the traces. The 'metadata' contains the plaintexts, keys, and masking randomness used during each encryption. The function returns the extracted traces and labels (pairs $(X\_profiling, Y\_profiling)$, (X_attack, Y_attack)), and optionally, the metadata, structured as tuples for further processing.

The ASCAD dataset had the labels for corresponding traces already stored as part of the dataset, and therefore, they are easy to load. However, for a better understanding, we also show how the labels can be created from the other information in the dataset. In Listing 9, we show how, by using the `metadata` information on the plaintexts and the keys, we can generate the correct labels. The same can be done to generate the /attack set labels (see the tutorial notebook).

Thus, plaintext and key information are part of the metadata property. It is common to perform side-channel analysis to retrieve a single key byte and repeat the process to recover the whole key, following a divide-and-conquer approach [7], [4], [42]. The 'target_byte' in the ASCAD case is commonly the third byte (in 0-index arrays, we write target_byte = 2) as it is the first masked key byte. The labels are the intermediate values in the execution of the algorithm, so the labels are calculated as $\texttt{S-box}[p_i \oplus key\_byte_i]$, where the AES `S-box` is a table of fixed and known values (see variable AES_Sbox in the tutorial), and p_i represents the plaintext corresponding to the i-th trace, while key_byte is the correct key_byte for that i-th trace. In our case, we use fixed-key ASCAD, so the key byte value is the same for all training data, but there is another version of the dataset where that is not the case, and the key varies between the traces. Thus, this code can also be used in that case without specific changes.

IV. Side-channel Traces Preprocessing

Once the side-channel measurements are loaded, standard preprocessing is usually performed before the training/profiling phase starts. Common preprocessing performed for deep learning in other domains is standardization and normalization.

Listing 10 performs standardization on our loaded input data ($X\_profiling$ and X_attack), and we need to make sure that the same transformation is done on both the profiling and attack traces, otherwise the trained model might have issues with the generalization. As this procedure is common, the `scikit-learn` (`sklearn`) library has the implementation of this method. Thus, we use the `StandardScaler` class to perform standardization, with `fit_transform` using the given data to first fit the parameters of the standardization, and then transforming it accordingly. On the other hand, we use a transform function on the attack dataset, as we no longer fit the data, but use the previous parameters to perform the same transformation on the attack set.

Listing 10. Performing standardizaion on loaded dataset.
```
scaler = StandardScaler()
X_profiling = scaler.fit_transform(X_profiling)
X_attack = scaler.transform(X_attack)
```

Similarly, `scikit-learn` has the `MinMaxScaler` that can be used for normalization as shown in Listing 11

Listing 11. Performing normalization on loaded dataset.
```
scaler = MinMaxScaler(feature_range=(0, 1))
X_profiling = scaler.fit_transform(X_profiling)
X_attack = scaler.transform(X_attack)
```

A. Feature Selection and Resampling

In SCA, crucial parts of the classical analysis are point of interest (POI) selection and feature engineering [65]. When traces contain many samples (for instance, already having 50 samples is computationally expensive when dealing with techniques that have cube complexity with respect to the number of features, as is the case of the template attack), running attacks directly on the full traces using classical (machine learning) methods becomes computationally prohibitive. Additionally, including many uninformative points can harm the performance of these methods significantly [66]. Thus, feature selection and feature engineering methods aim to extract features that hold more information for a more efficient or successful attack. With the use of deep learning algorithms, this step can be omitted, as DL is used to find representative features to learn a generalizable model for predictions on unseen data. However, while neural networks can be used to attack raw traces [67], using inputs with fewer features helps both in computation time and in the difficulty of finding appropriate parameters. These methods enable the use of smaller networks, faster attacks, and enable better explainability of the models. As such, in [63], the authors proposed to categorize POI selection into three distinct scenarios.

As we mentioned, in the ASCAD dataset that we use in the tutorial, while the raw traces are available, we use only a smaller subset of the samples from the traces. Namely, we use the time window of the raw traces that correspond to the leaking operation. We can do this as we have the full control over the profiling device. This is what is called **Optimized Points of Interest (OPOI)**, a setting that has been used most often in DLSCA research. Here, an attacker analyses traces and uses some knowledge about the implementation to select a range of points that they expect to contain the relevant leakage. The main benefits here are that the computational load and the number of uninformative points are more manageable. For ASCADf, this range contains 700 points over the 100 000 points in the raw traces (1 400 vs. 250 000 for ASCADr). In Listing 12, we show how to select the window for each trace.

Listing 12. OPOI window selection.
```
def load_ascad_window(ascad_database_file,
    load_metadata=False, target_byte=2, start_index
    =45400, end_index=46100):
  check_file_exists(ascad_database_file)
  # Open the ASCAD database HDF5 for reading
  in_file         = h5py.File(ascad_database_file, "r
      ")
  window_traces = in_file['traces'][:50000][
      start_index:end_index]
  window_attack = in_file['traces'][50000:][
      start_index:end_index]
```

In **Non-Optimized Points of Interest (NOPOI)** setting, an attacker directly mounts an end-to-end attack against raw traces. The first works targeting such long traces directly without preprocessing are [68], [67]. A key benefit over OPOI

Listing 9. Creating the labels for the ASCAD dataset.

```
(X_profiling, Y_profiling), (X_attack, Y_attack), (profiling_metadata, attack_metadata) = load_ascad(
    ascad_dataset, load_metadata=True, target_byte=target_byte)

# Load the plaintexts of the profiling traces of the corresponding target byte
profiling_plaintexts = profiling_metadata['plaintext'][:, target_byte].astype(np.uint8)
# Load the target_byte value of the key of the corresponding profiling traces
profiling_keys =  profiling_metadata['key'][:, target_byte].astype(np.uint8)
# Calculate the correct label using the plaintext and key byte of the profiling set
created_Y_profiling = AES_Sbox[profiling_plaintexts ^ profiling_keys]
```

is that additional leakage points may be outside of the considered interval, resulting in more efficient key extraction [67]. However, using 100 000 sample traces is still difficult and requires specialized architectures [67], [69]. To avoid dealing with such extremely long traces, we can resample each trace to include fewer points while still retaining important signal information. As shown in [63], models trained on resampled traces can often perform similarly efficient attacks to models trained on raw traces. In Listing 13, we provide an example of how to do the resampling. To resample, we take windows of 20 points, skipping 10 points between each point. This results in 10 times fewer samples, significantly improving both the memory requirements and the training time.

Listing 13. NOPOI trace-resampling.

```
# Resampling function
def winres(trace, window=20, overlap=0.5):
    trace_winres = []
    step = int(window * overlap)
    max = len(trace)
    for i in range(0, max, step):
        # Take average over 'window' sample points
        trace_winres.append(np.mean(trace[i:i +
            window]))
    return np.array(trace_winres)


window = 20
overlap = 0.5
original_trace_len = 100000
n_traces = 50000
new_trace_len = int(original_trace_len /(window*
    overlap))
resampled_traces = np.zeros((n_traces, new_trace_len
    ))
# Loop over traces to resample
for i in range(n_traces):
    resampled_traces[i] = winres(raw_traces[i],
        window, overlap)
```

Refined Points of Interest (RPOI) is a setting where an attacker selects only samples that contain relevant leakage information. Notably, when targeting masked implementations this requires access to masking randomness, which is not always a realistic assumption even in evaluation contexts with collaboration from developers [30]. Using this access, an evaluator can compute the first-order leakage of secret shares directly and select only the most relevant samples for further analysis. In Listing 14, we can see that we first define the function to compute Signal-to-Noise Ratio (SNR). This is a common function to measure the amount of leakage for a specific intermediate value at each point in the trace.

Listing 14. Signal-to-Noise Ratio computation.

```
# Function that calculates the Signal-to-Noise ratio
    (SNR).
def snr_fast(x, y):
    ns = x.shape[1]
    unique = np.unique(y)
    means = np.zeros((len(unique), ns))
    variances = np.zeros((len(unique), ns))

    # For each class, compute the mean and the
        variance at each index of trace
    for i, u in enumerate(unique):
        new_x = x[np.argwhere(y == int(u))]
        means[i] = np.mean(new_x, axis=0)
        variances[i] = np.var(new_x, axis=0)
    return np.var(means, axis=0) / np.mean(variances
        , axis=0)
```

Then, in Listing 15, we define a function that returns the indices with the n highest SNRs for a specific intermediate value. We utilize these to get the POIs for the two shares r_i and $\texttt{S-box}[p_i \oplus k_i] \oplus r_i$.[6]

Listing 15. RPOI feature selection.

```
masks = profiling_metadata["masks"][:, target_byte
    -2].astype(np.uint8)
masked_sbox_out = AES_Sbox[profiling_metadata["
    plaintext"][:, target_byte].astype(np.uint8) ^
    profiling_metadata["key"][:, target_byte].astype
    (np.uint8)] ^ masks

poi_sbox_out_masked = get_poi_share(X_profiling,
    masked_sbox_out)
poi_masks = get_poi_share(X_profiling, masks)
poi_full = np.append(poi_sbox_out_masked, poi_masks)

selected_features = X_profiling[:, poi_full]
selected_features_attack = X_attack[:, poi_full]
```

V. Profiling Phase

Profiling is the first phase of the profiling SCA, and the objective of this phase is to learn the mapping between the side-channel information and the correct key(s) of the cryptographic algorithm using the clone device of our target. Since the target is a clone, the assumption is that the model can generalize and be used to retrieve the correct key for the target device using the corresponding measurement and the obtained model from this phase.

A. Leakage Models and Label Preparation

The process for profiling is to collect a (large) set of profiling traces from the profiling device. We refer to this

[6]In the mask line, we have $target\_byte - 2$ as the first two masks are always zero and the ASCAD database authors did not include those in the fixed key version. This is not necessary for the variable key version, as these are included.

set of profiling traces as $\mathcal{X}_p$. This set $\mathcal{X}_p$ is an $n \times m$ array where n is the number of traces and m is the number of samples per trace. Each trace $X_i, 0 < i \leq n$ represents the side-channel measurement during one encryption/decryption. As for this profiling device, the key(s) is/are known, and a set of labels representing the targeted intermediate value(s) can be generated from the plaintexts and keys. Then, we can pick any specific intermediate value to attack.

For the ASCAD dataset we consider in this tutorial, the cipher is AES-128, and we generally target the output of the first round Substitution Box (`S-box`). This target is chosen as the output value depends on only one key byte and allows for efficient differential attacks during the attack phase [4]. The intermediate value for byte j for trace X_i is represented by:

$$IV_i = \texttt{S-box}[p_i^j \oplus k_i^j].$$

Then, the labels for the profiling set $\mathcal{Y}$ can be generated. For the i-th trace, the label Y_i is generated by using a *leakage model* that represents how we expect the intermediate value to physically leak in the side-channel trace. Common leakage models are the Hamming Weight (HW), i.e., the number of bits equal to 1 in the IV, the Identity (ID), i.e., directly using the IV, Hamming distance (HD), and a bitwise leakage model where we select a specific bit from the IV.[7]

Depending on the selected leakage model, we have a different number of possible labels for each trace, and therefore a number of classes, which dictate the number of neurons in the last output layer of the neural networks. Specifically, for the ID leakage model, we have a 256 possible label, while for Hamming Weight (or distance), we have *nine* classes. Note that if we want to use Hamming weight or other leakage models, we need to convert labels. The code that converts the ID values to HW is shown in Listing 16. The function `calculate_HW` takes the array of labels (data) as the input argument. The `bin(x).count("1")` calculates the Hamming weight by counting the number of ones in the string binary format of the given number x. Each label from the data array is then converted to the Hamming weight value. We can call this function on our loaded `Y_profiling` and `Y_attack` arrays like this: `Y_profiling = calculate_HW(Y_profiling)`.

Listing 16. Convert ID values to Hamming Weight.
```
def calculate_HW(data):
    hw = [bin(x).count("1") for x in range(256)]
    return [hw[int(s)] for s in data]
```

Once we have the labels in the desired leakage model, as we use classification, we convert the labels in one-hot encoded arrays, where depending on the number of labels, we have an array of that size with every value set to 0 except for the index with the correct label that is set to 1. For example, if we have Hamming weight, and therefore 9 classes $(0-8)$, the categorical one-hot encoded array of the class 4 that will be used by the neural network will be an array $[0,0,0,0,1,0,0,0,0]$. `TensorFlow` library has a function to convert the numerical values into the categorical classes and its use is shown in Listing 17.

Listing 17. Convert labels to categorical one-hot encoded arrays.
```
from tensorflow.keras.utils import to_categorical
Y_profiling = to_categorical(Y_profiling,
    num_classes=nb_classes)
Y_attack = to_categorical(Y_attack, num_classes=
    nb_classes)
```

B. Model Training

Once the data and labels are prepared, we can perform the model training as part of the profiling phase of SCA. The model is trained to learn the mapping between side-channel information (traces) and the correct intermediate values within the selected leakage model. We have already shown how models can be created with the `TensorFlow` library, specifically a simple MLP model. In the tutorial, there are several models one can test. These are models with fixed hyperparameters taken from other works, namely, mlp_best from [55], while the rest are from [63].

The functions already compile the model with given hyperparameters, so we can call the function to create and compile the model by using the functions as shown in Listing 18. Next, for the training, we use the fit function, which takes in the input x and the corresponding labels y arguments. Moreover, we can define the number of epochs and batch size we want to use for training. The batch size comes from the related work and is a return variable of the model function, while the number of epochs is set directly. Additionally, common practice is to shuffle the input pairs before each epoch, so the fit function offers the shuffle argument to the function as a Boolean flag.

Listing 18. Creating a model and training it.
```
model, batch_size = best_mlp_id_opoi_1400_ascadr(
    nb_classes, input_size)
nb_epochs = 100
history = model.fit(x=X_profiling, y=Y_profiling,
    shuffle=True, batch_size=batch_size, verbose=1,
    epochs=nb_epochs)
```

C. Model Validation

Recommended practice is to use a validation set to monitor over-fitting or utilize early-stopping. The validation set is a subset of the training data that is not used directly for training, making it the unseen data for the purpose of objective and unbiased evaluation of the model's performance. The library can again do this quite straightforwardly with the argument to the fit (training) function as shown in Listing 19. We can set the `validation_split` as a float value between 0 and 1 representing the fraction of the training data to be used as validation data. Instead of a fraction number, a separate dataset can be given to the `fit` function as the argument `validation_data` for the same purpose.

Listing 19. Use validation set.
```
history = model.fit(x=X_profiling, y=Y_profiling,
    validation_split=validation_split, shuffle=True,
     batch_size=batch_size, verbose=1, epochs=
    nb_epochs)
```

[7]Commonly, the Most/Least Significant Bits, (M/L)SB, are used in literature.

The `history` variable will hold a `History` object, where the `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values if applicable. The argument `verbose` is used to set how much information should be displayed during training (0 = silent, 1 = progress bar, 2 = one line per epoch). We set it to the *progress bar*, which displays additional information and can be used to track the training of the model. For more information or issues, the `TensorFlow` being a well-documented and maintained library, we direct the readers to the official documentation [48]. For the DLSCA purpose, the validation can also include the SCA metrics, such as guessing entropy/success rate, as visible in Figure III. For this, one can follow the steps we show in Section VI using the validation set and the known correct key.

D. Data augmentation

Data augmentation is a commonly used technique in DL that can be used to transform training examples using some predefined transformations to generate 'new' training examples. For example, in the image domain, we can zoom in, rotate, or crop images to generate additional training examples [70]. In SCA, the role of data augmentation is mainly to emulate the effects of hiding countermeasures to induce the neural network to be invariant to these transformations. Cagli et al. [29] first used custom data augmentations to attack traces that are distorted (i.e., misaligned) due to clock jitter during measurements. Overall, the two main data augmentation techniques used in practice are random shifts and additive noise [71]. In Listing 20, we show examples of implementing these techniques to generate new traces. For both functions, we first generate a copy of the original traces. Then, we use the `np.tile` function to repeat the data n times. Finally, we take the repeated traces and transform them using either additive noise or random shifts. For the additive noise, it is straightforward to generate and add noise using standard numpy functionality to generate noise according to a normal distribution. For random shifts, we need to shift traces for a random number of samples. For each trace to augment, we first generate a random integer in [-max_shift, max_shift]. Subsequently, we use the numpy roll function to shift the trace by the number of points we generated. Note that when points are shifted past the end/start of the trace, they roll around to the start/end.

Listing 20. Data Augmentation functions

```
# Adding Gaussian noise
def data_augmentation_noise(orig_x_prof, num_repeats
    =2, std=0.01):
  # create copy of original
  new_x_prof = orig_x_prof.copy()
  nb_orig_traces = len(orig_x_prof)

  # repeat it n times
  new_x_prof = np.tile(new_x_prof, (num_repeats, 1))

  # add gaussian noise to added traces (original
      traces remain without additional noise)
  new_x_prof[nb_orig_traces:] += np.random.normal(0,
      std, new_x_prof[nb_orig_traces:].shape)
  return new_x_prof

# Adding random shifts
def data_augmentation_shift(orig_x_prof, num_repeats
    =2, max_shift=10):
  # create copy of original
  new_x_prof = orig_x_prof.copy()
  nb_orig_traces = len(orig_x_prof)

  # repeat it n times
  new_x_prof = np.tile(new_x_prof, (num_repeats, 1))

  # shift each trace randomly using np roll function
  for i in range(len(new_x_prof[nb_orig_traces:])):
    shift = random.randint(-max_shift, max_shift)
    new_x_prof[nb_orig_traces+i] = np.roll(
        new_x_prof[nb_orig_traces+i], shift)
  return new_x_prof
```

In Listing 21, we show how to use the functions described in Listing 20 to train a model. We first limit the number of (original) profiling traces to 50 000 to emulate settings where we have insufficient data to fit a neural network. Then, we define the number of repeats as 2 to generate an additional 50 000 traces every epoch. We use the tile function for the labels `Y_prof` to match the number of training examples in the augmented training set. Finally, in every epoch, we randomly generate the additional traces using the data augmentation function for random shifts and fit the model for one epoch on this augmented dataset.

Listing 21. Data Augmentation training.

```
nb_epochs = 100
n_repeats = 2
nb_prof_traces = 50000
X_prof = X_profiling[:nb_prof_traces]
Y_prof = Y_profiling[:nb_prof_traces]

# Define model
model, batch_size = best_mlp_id_opoi_1400_ascadr(
    nb_classes, input_size)

# Repeat the data n times using tile
Y_prof = np.tile(Y_prof, (n_repeats, 1))

for i in range(nb_epochs):
  new_x = data_augmentation_shift(X_prof,
      num_repeats=n_repeats)
  model.fit(x=new_x, y=Y_prof, shuffle=True,
      batch_size=batch_size, verbose=1, epochs=1)
```

E. Hyperparameter Tuning

In the previous step, we used fixed neural network architectures from related work. However, finding an appropriate model architecture for a specific target is often difficult. While in other deep learning domains, pretrained models with relatively standard training recipes are the norm, in DLSCA, every target has different characteristics requiring target-specific adaptations. Methodologies or general architectures have been proposed [72], [69], but directly applying these to novel datasets often still requires tuning of hyperparameters. Automated hyperparameter tuning is one common approach for avoiding reliance on human expertise in finding appropriate model configurations. The key idea is to train a large number of models and to evaluate these on a validation set to assess their effectiveness. In Listing 22, we show an example of running a random search strategy to train multiple models. The `mlp_random` function generates a random MLP model

from a pre-defined range of hyperparameters. Each model is then trained and subsequently evaluated using the validation set. We will explain the evaluation in Section VI in detail, but we mention it here, as the evaluation results and corresponding hyperparameter configurations and model weights are then stored in a list such that we can later retrieve the best model. Note that while in this example we run the model trainings/attacks consecutively, these can also be run in parallel on separate machines as the individual model trainings are independent. In the tutorial notebook, we also provide code for generating random CNN model configurations.

Listing 22. Hyperparameter tuning.
```
for i in range(nb_models):
    model, hp = mlp_random(classes=nb_classes,
        input_size=input_size)
    model.fit(x=X_profiling[:nb_prof_traces], y=
        Y_profiling[:nb_prof_traces], batch_size=hp[
        "batch_size"], verbose=0, epochs=
        nb_epochs_per_model, shuffle=True)
    predictions = model.predict(X_val)
    ge_median, ge_avg = guessing_entropy(predictions
        , validation_pt, correct_key_val, 200)
    avg_ge_list.append(ge_avg)
    model_list.append(model.get_weights())
    hp_list.append(hp)
```

Note that the function mlp_random randomly selects the hyperparameters from the range defined similarly to Listing 23. While creating the model, the function will perform steps as shown in Listing 24. Note that both of these listings show only parts of what is in the tutorial notebook and the full implementation in the notebook.

Listing 23. Setting the random search for hyperparameter tuning.
```
"batch_size": random.randrange(100, 1100, 100),
"layers": random.randrange(1, max_dense_layers + 1,
    1),
"neurons": random.choice([10, 20, 50, 100, 200, 300,
     400, 500]),
"activation": random.choice(["relu", "selu"])
```

Listing 24. Creating a random MLP using randomly selected hyperparameters.
```
for layer_index in range(hp["layers"]):
    x = Dense(hp["neurons"], activation=hp["
        activation"], kernel_initializer=hp["
        kernel_initializer"], name='dense_{}'.format
        (layer_index))(inputs if layer_index == 0
        else x)
```

For the current public targets, random search (with appropriate hyperparameter ranges) provides satisfactory performance [63]. However, its performance relies on the presence of successful models in the pre-defined search space. As such, several works have proposed more complex neural architecture search techniques that can find successful models without relying on restricted ranges. The main examples use reinforcement learning [28], Bayesian optimization [73], and neuroevolution [74].

F. Ensembles

Another option for optimizing attack performance is to use ensembles [75]. Ensemble learning trains two or more machine learning algorithms on a specific task and combines these models into a better-performing model. In our ensemble example, we combine the outputs of several models from the hyperparameter search as shown in Listing 25. We first define the number of models to use and sort the models from the hyperparameter search based on their validation performance. Then, for each of the top models, we recreate the trained model using the stored hyperparameters and trained weights. We then add the predictions of each of the top models and finally divide by the number of models to find the average. This is a simple and straightforward way to combine the results of different models. However, ensembles can, in some cases, dictate how the training is done as well (e.g., Bootstrap aggregation (bagging), boosting), and how the models are combined (e.g., voting majority, averaging) [76]. For instance, in [77], the authors explore more advanced combination methods to improve ensemble learning for profiling SCA.

Listing 25. Ensemble.
```
# The number of models to use for the ensemble
nb_models_ensemble = 5

# Sort model indices based on GE
sorted_model_indices = np.argsort(avg_ge_list)

# Take predictions from top N models
predictions_tot = np.zeros((X_attack.shape[0],
    nb_classes))

for i in range(nb_models_ensemble):
  model_num = sorted_model_indices[i]
  # Recreate model based on hyperparameters
  hp = hp_list[model_num]
  model, _ = mlp_random(classes=nb_classes,
      input_size=input_size, hp=hp)

  # Restore trained weights
  model.set_weights(model_list[model_num])

  # Add to total predictions
  predictions_tot += model.predict(X_attack)

predictions_tot = predictions_tot/nb_models_ensemble
```

VI. Attack and Evaluation

During model training, we rely on optimization procedures for standard loss functions in machine learning. However, we want to evaluate the profiling phase with metrics that measure how well attacks perform in practice. We already show that the SCA metrics can be used on the validation set to evaluate the performance of the model. Our next step will be performing the attack and evaluating how well the trained model works on the attack dataset. Note that the attack set is different from the validation set.

The main reason for using SCA-specific metrics is that common ML metrics like accuracy and loss are not good predictors of attack performance [78]. While models that achieve very high accuracies lead to efficient key retrieval, model accuracies close to random guessing do not imply we cannot retrieve the key. Indeed, by accumulating predictions across a larger number of traces, we can score key candidates and rank them to extract the most likely candidate. If the model is trained well, the estimated most likely key will then be (one of) the top candidates(s). In the tutorial, we provide the code for calculating the guessing entropy, a common metric that

estimates the average rank of the correct key candidate by conducting several simulated attacks on the attack set. The success rate can be implemented similarly.

First, we do not use the top predicted label but the predictions that hold a probability distribution over all possible classes for each attack trace. Thus, the predictions variable from Listing 4 has dimension $(N, nb\_classes)$, where N is the number of traces in the dataset (in this case, the attack dataset), and $nb\_classes$ is the number of possible classes (neurons in the output layer). Since we will have to accumulate the probabilities across a large number of traces, we use the log of the probability values, as multiplying probabilities across several traces can be problematic. Additionally, we add a small value to the original predicted probabilities for numerical stability while taking the log of values close to zero. This is shown in Listing 26.

Listing 26. Calculate the log probabilities.
```
predictions_log = np.log(predictions + 1e-36)
```

Given the predictions in log values and plaintexts corresponding to each trace in the attack dataset, we calculate the probability for all key byte hypotheses of how likely that key byte hypothesis is to be the correct key. As we are trying to retrieve one byte of the first round key, which is a value of 8 bits, we have 256 possible key values (for an `S-box` of 8 bits). For each key hypothesis, we calculate the hypothetical labels if the correct key would be that specific key hypothesis. We calculate it using the public AES `S-box` and known plaintexts. Depending on the leakage model, we use the value directly or apply the appropriate conversion to the desired leakage model. This process is shown in the first part of Listing 27. Since we have the hypothetical labels for each key, we can now score each key hypothesis by summing the log predictions of the hypothetical labels over the whole attack set, which is shown in the second part of Listing 27. In the tutorial, this is enclosed in the function called score_keys.

Listing 27. Calculate the probabilities for all key byte hypotheses.
```
scores_keys = np.zeros(256)
for k in range(256):
  #Generate Hypothetical labels for a key candidate
      k
  hypothetical_labels = AES_Sbox[plaintexts ^ k]
  if leakage_model == "HW":
    hypothetical_labels = calculate_HW(
        hypothetical_labels)

  for i in range(predictions.shape[0]):
    scores_keys[k] += predictions[i,
        hypothetical_labels[i]]
```

In the case of an actual attack, this is where the attacker would use the most likely key byte from the calculations to attack and try to encrypt/decrypt the information. Concretely, an attacker would attack each key byte separately to obtain rankings and then use this information to enumerate keys more efficiently. In the case of AES with 16 key bytes, if the correct candidate is always in the top-10 for each byte, the full correct key will be in the first 10^{16} guesses. This significantly improves over enumeration without incorporating SCA where we have 2^{8*16} possible keys.

In the context of research on DLSCA, we use public datasets where the correct key is known, even for the attack set. Since our goal is to evaluate the performance of the models and proposed methods, we assess the attack using SCA metrics and the known correct key byte. Moreover, in the context of a security evaluation, we will know the correct key even for the device(s) that are being 'attacked'. Using the information about the correct key, key rank is calculated as shown in Listing 28. The key_scores are the log probabilities for each key hypothesis obtained with the explained score_keys function. The key rank is calculated by sorting the key hypotheses in descending order based on their scores and retrieving the index of the correct key. Similarly, for success rate, we can check whether the key rank is within the first o key candidates in sorted keys and return 1 if it is or 0 when it is not. As mentioned above in Section V-C, we can perform this evaluation on the validation set, which is part of the training set (profiling device) for which we know the key(s). Here, the evaluation is used for the selection of the best model to use for the actual attack, where the most likely key will be selected based on the key hypothesis scoring (function score_keys).

Listing 28. Key rank.
```
key_scores = score_keys(predictions_log, plaintexts)
order_keys = np.argsort(key_scores)[::-1]
key_rank, = np.where(order_keys == correct_key)
```

Using the code in Listing 28, we can compute the rank of the correct key. However, guessing entropy is the average key rank, so we need to simulate multiple attacks and evaluate the average key rank empirically. The function guessing_entropy in the tutorial notebook implements this process and is shown in Listing 29. Instead of using the complete attack dataset, we use a random subset of traces to calculate the key rank. The key ranks for each attack are then averaged to obtain the guessing entropy. The median can also be used [79], so the function returns both. The same process can be followed to simulate a number of attacks and compute the success rate, i.e., the percentage of attacks where the key rank is 0.

Listing 29. Guessing entropy.
```
def guessing_entropy(predictions, plaintexts,
    correct_key, nb_traces, nb_attacks=100):
  ranks = np.zeros(nb_attacks)
  # Take the log of probabilities to sum later with
      a small addition for numeric stability
  predictions_log = np.log(predictions + 1e-36)

  for attack in range(nb_attacks):
    # Take random subset of traces
    r = np.random.choice(
            range(predictions_log.shape[0]),
                nb_traces, replace=False)

    key_scores = score_keys(predictions_log[r],
        plaintexts[r])

    order_keys = np.argsort(key_scores)[::-1]

    ranks[attack], = np.where(order_keys ==
        correct_key)
  return np.median(ranks), np.average(ranks)
```

In SCA, we also care about the number of attack traces that are needed to extract the correct key, which is when

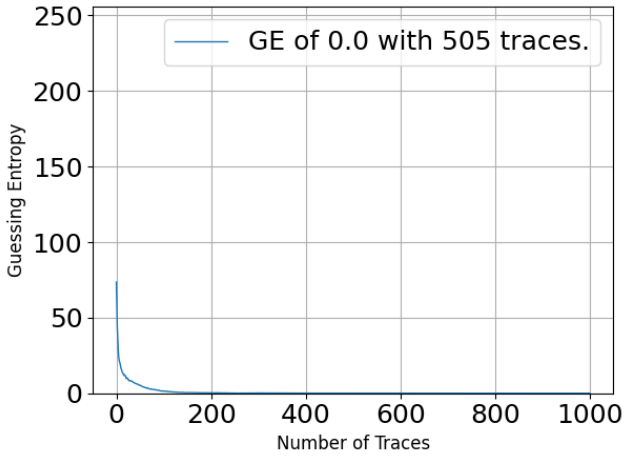

Fig. 3. GE results for increasing number of traces using the model trained above.

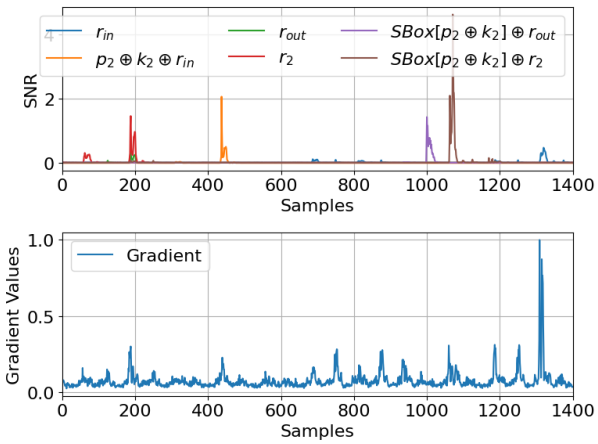

Fig. 4. SNR and gradient visualization for network trained in Section V-B.

the guessing entropy equals 0, indicating that the correct key candidate is always ranked first. Thus, we can call the function guessing_entropy with argument nb_traces set to 1, then 2, etc. Running that function multiple times with an increasing number of attacks would be quite expensive. Thus, we provide a more common implementation of GE where this is already done within the function itself for more optimized computation (see function guessing_entropy_convergence and score_keys_convergence in the tutorial notebook). As is common in the related literature, we show the progression of GE using increasing traces in Figure 3. This plot showcases that the GE decreases as predictions from additional traces are included in the key scoring.

VII. Explainability

When a model is found that can successfully retrieve the key, it might be useful to understand what leakage the model is exploiting. Furthermore, a better understanding of *how* models break a target might enable future improvements to attack methods or defense mechanisms. With this in mind, the research community is interested in explainability approaches in SCA.

In Listing 30, we show an example of an input visualization technique that shows what samples in the input impact the model predictions. The gradient-based technique uses back-propagation to differentiate the loss with respect to each of the sample points and was introduced for SCA in [80]. In Figure 4, we show how the resulting graph relates to the SNRs of various secret shares and the influence of each trace point on the model outputs. Overall, we can see that higher gradient values align with samples that show leakage in the SNR plot. These plots can then be used to determine which shares contribute to the leakage the model exploits and the corresponding locations in the trace where those shares leak.

Listing 30. Gradient Visualization.

```
def gradient_vis(traces, labels):
    x_tf = tf.Variable(traces)
    with tf.GradientTape() as tape:
        tape.watch(x_tf)
        pred = model(x_tf, training=False)
        loss = tf.keras.losses.CategoricalCrossentropy()
            (labels, pred)

    grads = tape.gradient(loss, x_tf)
    dgrad_abs = tf.math.abs(grads)

    return np.average(dgrad_abs.numpy(), axis=0)
```

Several other input attribution techniques have been used in the SCA context. An overview of these can be found in [81]. Furthermore, occlusion-based methods that mask out certain parts of the input to determine which parts are necessary for model performance can also be used for this purpose [82], [83].

Some approaches have also attempted to gain a better understanding of the internal behavior of trained networks. In [84], the similarity of networks trained in multi-device settings and across different datasets. In [85], the authors proposed using a more interpretable type of network architecture. Finally, in [86], the authors used ablations to evaluate which layers in the model are most important to defeat certain countermeasures.

VIII. Conclusions

This tutorial examines deep learning-based side-channel analysis and provides steps for performing such analysis using publicly available datasets. The tutorial is supported by the provided Colab notebook, and explanations of the code are given in this article, accompanied by the relevant code snippets. We cover the fundamental DLSCA methods. Moreover, we point the interested readers to the more advanced methods with references within the corresponding step of the DLSCA approach.

As the AIS 46 makes a step toward standardization of DLSCA, there might be more interest in DLSCA, making this tutorial a relevant introduction to the field. We consider this tutorial helpful for anyone interested in understanding how deep learning can exploit side-channel leakages, learn practical techniques to perform DLSCA, and improve the security of hardware and cryptographic implementations.

ACKNOWLEDGMENT

This work was (in part) supported by the Dutch Research Council (NWO) through the Challenges in Cyber Security (CiCS) project of the Gravitation research program under the grant 024.006.037.

REFERENCES

[1] C. Paar and J. Pelzl, *Understanding cryptography*. Springer, 2010, vol. 1.

[2] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007.

[3] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[4] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31.

[5] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523. Springer, 2002, pp. 13–28. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_3

[6] I. J. S. 27. (2022) Iso/iec 15408-1:2022: Information security, cybersecurity and privacy protection - evaluation criteria for it security. [Online]. Available: https://www.iso.org/standard/72891.html

[7] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, ser. Lecture Notes in Computer Science, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 388–397. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_25

[8] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *International Conference on Research in Smart Cards*. Springer, 2001, pp. 200–210.

[9] D. Genkin, A. Shamir, and E. Tromer, "Acoustic cryptanalysis," *Journal of Cryptology*, vol. 30, pp. 392–443, 2017.

[10] G. Hospodar, B. Gierlichs, E. D. Mulder, I. Verbauwhede, and J. Vandewalle, "Machine learning in side-channel analysis: a first study," *J. Cryptogr. Eng.*, vol. 1, no. 4, pp. 293–302, 2011. [Online]. Available: https://doi.org/10.1007/s13389-011-0023-x

[11] A. Heuser and M. Zohner, "Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines," in *COSADE*, ser. LNCS, W. Schindler and S. A. Huss, Eds., vol. 7275. Springer, 2012, pp. 249–264.

[12] L. Lerman, G. Bontempi, and O. Markowitch, "Side channel attack: an approach based on machine learning," *Center for Advanced Security Research Darmstadt*, vol. 29, pp. 29–41, 2011.

[13] Z. Martinasek and V. Zeman, "Innovative method of the power analysis," *Radioengineering*, vol. 22, no. 2, pp. 586–594, 2013.

[14] Z. Martinasek, P. Dzurenda, and L. Malina, "Profiling power analysis attack based on mlp in dpa contest v4. 2," in *2016 39th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, 2016, pp. 223–226.

[15] H. Maghrebi, T. Portigliatti, and E. Prouff, "Breaking cryptographic implementations using deep learning techniques," in *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, ser. Lecture Notes in Computer Science, C. Carlet, M. A. Hasan, and V. Saraswat, Eds., vol. 10076. Springer, 2016, pp. 3–26. [Online]. Available: https://doi.org/10.1007/978-3-319-49445-6_1

[16] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina, "Sok: Deep learning-based physical side-channel analysis," *ACM Comput. Surv.*, vol. 55, no. 11, Feb. 2023. [Online]. Available: https://doi.org/10.1145/3569577

[17] Federal Office for Information Security (BSI), "Guidelines for Evaluating Machine-Learning based Side-Channel Attack Resistance," https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_AI_guide.pdf?__blob=publicationFile&v=6, 02 2024, technical Report AIS 46.

[18] L. Masure, C. Dumas, and E. Prouff, "A comprehensive study of deep learning for side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, pp. 348–375, Nov. 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8402

[19] S. Picek, "Challenges in deep learning-based profiled side-channel analysis," in *Security, Privacy, and Applied Cryptography Engineering - 9th International Conference, SPACE 2019, Gandhinagar, India, December 3-7, 2019, Proceedings*, ser. Lecture Notes in Computer Science, S. Bhasin, A. Mendelson, and M. Nandi, Eds., vol. 11947. Springer, 2019, pp. 9–12. [Online]. Available: https://doi.org/10.1007/978-3-030-35869-3_3

[20] B. Hettwer, S. Gehrer, and T. Güneysu, "Applications of machine learning techniques in side-channel attacks: a survey," *J. Cryptogr. Eng.*, vol. 10, no. 2, pp. 135–162, 2020. [Online]. Available: https://doi.org/10.1007/s13389-019-00212-8

[21] S. Jin, S. Kim, H. Kim, and S. Hong, "Recent advances in deep learning-based side-channel analysis," *Etri Journal*, vol. 42, no. 2, pp. 292–304, 2020.

[22] L. Batina, M. Djukanovic, A. Heuser, and S. Picek, "It started with templates: The future of profiling in side-channel analysis," in *Security of Ubiquitous Computing Systems*, G. Avoine and J. Hernandez-Castro, Eds. Springer International Publishing, 2021, pp. 133–145. [Online]. Available: https://doi.org/10.1007/978-3-030-10591-4_8

[23] M. Krcek, H. Li, S. Paguada, U. Rioja, L. Wu, G. Perin, and L. Chmielewski, "Deep learning on side-channel analysis," in *Security and Artificial Intelligence*, 2022, pp. 48–71. [Online]. Available: https://doi.org/10.1007/978-3-030-98795-4_3

[24] G. Perin, L. Wu, and S. Picek, "Aisy-deep learning-based framework for side-channel analysis," *Cryptology ePrint Archive*, 2021.

[25] E. Bursztein, L. Invernizzi, K. Král, and J.-M. Picod, "SCAAML: Side Channel Attacks Assisted with Machine Learning," 2019. [Online]. Available: https://github.com/google/scaaml

[26] M. Brisfors and S. Forsmark, "DLSCA: a tool for deep learning side channel analysis," Cryptology ePrint Archive, Paper 2019/1071, 2019. [Online]. Available: https://eprint.iacr.org/2019/1071

[27] F. Koeune and F.-X. Standaert, "A Tutorial on Physical Security and Side-Channel Attacks," in *Foundations of Security Analysis and Design III: FOSAD 2004/2005 Tutorial Lectures*, A. Aldini, R. Gorrieri, and F. Martinelli, Eds. Berlin, Heidelberg: Springer, 2005, pp. 78–108. [Online]. Available: https://doi.org/10.1007/11554578_3

[28] J. Rijsdijk, L. Wu, G. Perin, and S. Picek, "Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 3, pp. 677–707, 2021. [Online]. Available: https://doi.org/10.46586/tches.v2021.i3.677-707

[29] E. Cagli, C. Dumas, and E. Prouff, "Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing," in *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, ser. Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Springer, 2017, pp. 45–68. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_3

[30] L. Masure, V. Cristiani, M. Lecomte, and F. Standaert, "Don't learn what you already know scheme-aware modeling for profiling side-channel analysis against masking," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2023, no. 1, pp. 32–59, 2023. [Online]. Available: https://doi.org/10.46586/tches.v2023.i1.32-59

[31] S. Karayalcin, M. Krcek, L. Wu, S. Picek, and G. Perin, "It's a kind of magic: A novel conditional GAN framework for efficient profiling side-channel analysis," in *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part VIII*, ser. Lecture Notes in Computer Science, K. Chung and Y. Sasaki, Eds., vol. 15491. Springer, 2024, pp. 99–131. [Online]. Available: https://doi.org/10.1007/978-981-96-0944-4_4

[32] M. Staib and A. Moradi, "Deep learning side-channel collision attack," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 3, p. 422–444, Jun. 2023. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/10969

[33] "Google Colab." [Online]. Available: https://colab.research.google.com/

[34] "Project Jupyter." [Online]. Available: https://jupyter.org

[35] J. Daemen and V. Rijmen, "Reijndael: The advanced encryption standard." *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 26, no. 3, pp. 137–139, 2001.

[36] 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, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 443–461.

[37] O. Bronchain, J. M. Hendrickx, C. Massart, A. Olshevsky, and F. Standaert, "Leakage certification revisited: Bounding model errors in

side-channel security evaluations," *IACR Cryptol. ePrint Arch.*, p. 132, 2019. [Online]. Available: https://eprint.iacr.org/2019/132

[38] A. Ito, R. Ueno, and N. Homma, "Perceived information revisited: New metrics to evaluate success rate of side-channel attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, p. 228–254, Aug. 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9819

[39] ——, "Perceived information revisited ii: Information-theoretical analysis of deep-learning based side-channel attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, p. 450–474, Dec. 2024. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11936

[40] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, ser. Lecture Notes in Computer Science, M. Joye and J. Quisquater, Eds., vol. 3156. Springer, 2004, pp. 16–29. [Online]. Available: https://doi.org/10.1007/978-3-540-28632-5_2

[41] Y. Ishai, A. Sahai, and D. A. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, ser. Lecture Notes in Computer Science, D. Boneh, Ed., vol. 2729. Springer, 2003, pp. 463–481. [Online]. Available: https://doi.org/10.1007/978-3-540-45146-4_27

[42] F.-X. Standaert, *Side-Channel Analysis and Leakage-Resistance*. Version 1.2, September 2024.

[43] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[44] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[45] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009, vol. 2.

[46] S. Sra, S. Nowozin, and S. J. Wright, *Optimization for machine learning*. MIT press, 2011.

[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[48] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[49] B. Timon, "Non-profiled deep learning-based side-channel attacks with sensitivity analysis," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 2, pp. 107–131, 2019. [Online]. Available: https://doi.org/10.13154/tches.v2019.i2.107-131

[50] V.-P. Hoang, N.-T. Do, and V. S. Doan, "Efficient non-profiled side channel attack using multi-output classification neural network," *IEEE Embedded Systems Letters*, pp. 1–1, 2022.

[51] N.-T. Do, P.-C. Le, V.-P. Hoang, V.-S. Doan, H. G. Nguyen, and C.-K. Pham, "Mo-dlsca: Deep learning based non-profiled side channel analysis using multi-output neural networks," in *2022 International Conference on Advanced Technologies for Communications (ATC)*, 2022, pp. 245–250.

[52] N.-T. Do, V.-P. Hoang, and V. S. Doan, "A novel non-profiled side channel attack based on multi-output regression neural network," *Journal of Cryptographic Engineering*, pp. 1–13, 2023.

[53] I. Savu, M. Krček, G. Perin, L. Wu, and S. Picek, "The need for more: unsupervised side-channel analysis with single network training and multi-output regression," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2024, pp. 113–132.

[54] L. Wu and S. Picek, "Remove some noise: On pre-processing of side-channel measurements with autoencoders," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 389–415, Aug. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8688

[55] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ASCAD database," *J. Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, 2020. [Online]. Available: https://doi.org/10.1007/s13389-019-00220-8

[56] S. Bhasin, A. Chattopadhyay, A. Heuser, D. Jap, S. Picek, and R. R. Shrivastwa, "Mind the portability: A warriors guide through realistic profiled side-channel analysis," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss2020/

[57] L. Wu, G. Perin, and S. Picek, "Weakly profiling side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 3, p. 707–730, Nov. 2024. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11901

[58] A. Rezaeezade, T. Yap, D. Jap, S. Bhasin, and S. Picek, "Breaking the blindfold: Deep learning-based blind side-channel analysis," Cryptology ePrint Archive, Paper 2025/157, 2025. [Online]. Available: https://eprint.iacr.org/2025/157

[59] J. Van Woudenberg and C. O'Flynn, *The hardware hacking handbook: breaking embedded security with hardware attacks*. No Starch Press, 2021.

[60] F.-X. Standaert, "Introduction to side-channel attacks," *Secure integrated circuits and systems*, pp. 27–42, 2010.

[61] A. Krizhevsky, "Learning multiple layers of features from tiny images," pp. 32–33, 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[62] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[63] G. Perin, L. Wu, and S. Picek, "Exploring feature selection scenarios for deep learning-based side-channel analysis," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 4, pp. 828–861, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i4.828-861

[64] L. Masure and R. Strullu, "Side-channel analysis against anssi's protected AES implementation on ARM: end-to-end attacks with multi-task learning," *J. Cryptogr. Eng.*, vol. 13, no. 2, pp. 129–147, 2023. [Online]. Available: https://doi.org/10.1007/s13389-023-00311-7

[65] E. Cagli, "Feature extraction for side-channel attacks. (extraction de caractéristiques pour les attaques par canaux auxiliaires)," Ph.D. dissertation, Sorbonne University, France, 2018. [Online]. Available: https://tel.archives-ouvertes.fr/tel-02494260

[66] L. Lerman, R. Poussier, O. Markowitch, and F. Standaert, "Template attacks versus machine learning revisited and the curse of dimensionality in side-channel analysis: extended version," *J. Cryptogr. Eng.*, vol. 8, no. 4, pp. 301–313, 2018. [Online]. Available: https://doi.org/10.1007/s13389-017-0162-9

[67] X. Lu, C. Zhang, P. Cao, D. Gu, and H. Lu, "Pay attention to raw traces: A deep learning architecture for end-to-end profiling attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, p. 235–274, Jul. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8974

[68] L. Masure, N. Belleville, E. Cagli, M. Cornelie, D. Couroussé, C. Dumas, and L. Maingault, "Deep learning side-channel analysis on large-scale traces - A case study on a polymorphic AES," in *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, L. Chen, N. Li, K. Liang, and S. A. Schneider, Eds., vol. 12308. Springer, 2020, pp. 440–460. [Online]. Available: https://doi.org/10.1007/978-3-030-58951-6_22

[69] E. Bursztein, L. Invernizzi, K. Král, D. Moghimi, J. Picod, and M. Zhang, "Generalized power attacks against crypto hardware using long-range deep learning," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2024, no. 3, pp. 472–499, 2024. [Online]. Available: https://doi.org/10.46586/tches.v2024.i3.472-499

[70] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[71] H. Li and G. Perin, "A systematic study of data augmentation for protected AES implementations," *J. Cryptogr. Eng.*, vol. 14, no. 4, pp. 649–666, 2024. [Online]. Available: https://doi.org/10.1007/s13389-024-00363-3

[72] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, "Methodology for efficient CNN architectures in profiling attacks," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 1, pp. 1–36, 2020. [Online]. Available: https://doi.org/10.13154/tches.v2020.i1.1-36

[73] L. Wu, G. Perin, and S. Picek, "I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis,"

IEEE Trans. Emerg. Top. Comput., vol. 12, no. 2, pp. 546–557, 2024. [Online]. Available: https://doi.org/10.1109/TETC.2022.3218372

[74] R. Y. Acharya, F. Ganji, and D. Forte, "Information theory-based evolution of neural networks for side-channel analysis," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2023, no. 1, pp. 401–437, 2023. [Online]. Available: https://doi.org/10.46586/tches.v2023.i1.401-437

[75] G. Perin, L. Chmielewski, and S. Picek, "Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 337–364, Aug. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8686

[76] Y. Yang, H. Lv, and N. Chen, "A survey on ensemble learning under the era of deep learning," *Artif. Intell. Rev.*, vol. 56, no. 6, pp. 5545–5589, 2023. [Online]. Available: https://doi.org/10.1007/s10462-022-10283-5

[77] D. Llavata, E. Cagli, R. Eyraud, V. Grosso, and L. Bossuet, "Deep stacking ensemble learning applied to profiling side-channel attacks," in *Smart Card Research and Advanced Applications - 22nd International Conference, CARDIS 2023, Amsterdam, The Netherlands, November 14-16, 2023, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Bhasin and T. Roche, Eds., vol. 14530. Springer, 2023, pp. 235–255. [Online]. Available: https://doi.org/10.1007/978-3-031-54409-5_12

[78] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni, "The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 1, pp. 1–29, 2019.

[79] L. Wu, G. Perin, and S. Picek, "On the evaluation of deep learning-based side-channel analysis," in *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. Balasch and C. O'Flynn, Eds., vol. 13211. Springer, 2022, pp. 49–71. [Online]. Available: https://doi.org/10.1007/978-3-030-99766-3_3

[80] L. Masure, C. Dumas, and E. Prouff, "Gradient visualization for general characterization in profiling attacks," in *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, ser. Lecture Notes in Computer Science, I. Polian and M. Stöttinger, Eds., vol. 11421. Springer, 2019, pp. 145–167. [Online]. Available: https://doi.org/10.1007/978-3-030-16350-1_9

[81] B. Hettwer, S. Gehrer, and T. Güneysu, "Deep neural network attribution methods for leakage analysis and symmetric key recovery," in *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, K. G. Paterson and D. Stebila, Eds., vol. 11959. Springer, 2019, pp. 645–666. [Online]. Available: https://doi.org/10.1007/978-3-030-38471-5_26

[82] T. Schamberger, M. Egger, and L. Tebelmann, "Hide and seek: Using occlusion techniques for side-channel leakage attribution in cnns - an evaluation of the ASCAD databases," in *Applied Cryptography and Network Security Workshops - ACNS 2023 Satellite Workshops, ADSC, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Kyoto, Japan, June 19-22, 2023, Proceedings*, ser. Lecture Notes in Computer Science, J. Zhou, L. Batina, Z. Li, J. Lin, E. Losiouk, S. Majumdar, D. Mashima, W. Meng, S. Picek, M. A. Rahman, J. Shao, M. Shimaoka, E. O. Soremekun, C. Su, J. S. Teh, A. Udovenko, C. Wang, L. Y. Zhang, and Y. Zhauniarovich, Eds., vol. 13907. Springer, 2023, pp. 139–158. [Online]. Available: https://doi.org/10.1007/978-3-031-41181-6_8

[83] T. Yap, S. Bhasin, and S. Picek, "Occpois: Points of interest based on neural network's key recovery in side-channel analysis through occlusion," *IACR Cryptol. ePrint Arch.*, p. 1055, 2023. [Online]. Available: https://eprint.iacr.org/2023/1055

[84] D. van der Valk, S. Picek, and S. Bhasin, "Kilroy was here: The first step towards explainability of neural networks in profiled side-channel analysis," in *Constructive Side-Channel Analysis and Secure Design - 11th International Workshop, COSADE 2020, Lugano, Switzerland, April 1-3, 2020, Revised Selected Papers*, ser. Lecture Notes in Computer Science, G. M. Bertoni and F. Regazzoni, Eds., vol. 12244. Springer, 2020, pp. 175–199. [Online]. Available: https://doi.org/10.1007/978-3-030-68773-1_9

[85] T. Yap, A. Benamira, S. Bhasin, and T. Peyrin, "Peek into the black-box: Interpretable neural network using SAT equations in side-channel analysis," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2023, no. 2, pp. 24–53, 2023. [Online]. Available: https://doi.org/10.46586/tches.v2023.i2.24-53

[86] L. Wu, Y. Won, D. Jap, G. Perin, S. Bhasin, and S. Picek, "Ablation analysis for multi-device deep learning-based physical side-channel analysis," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 3, pp. 1331–1341, 2024. [Online]. Available: https://doi.org/10.1109/TDSC.2023.3278857