AISY - Deep Learning-based Framework for Side-channel Analysis

Guilherme Perin, Lichao Wu, Stjepan Picek

Cyber Security Research Group, Delft University of Technology,
Mekelweg 2, Delft, The Netherlands
`g.perin, l.wu-4, s.picek@tudelft.nl`

Abstract. The deep learning-based side-channel analysis represents an active research domain. While it is clear that deep learning enables powerful side-channel attacks, the variety of research scenarios often makes the results difficult to reproduce.
In this paper, we present AISY - a deep learning-based framework for profiling side-channel analysis. Our framework enables the users to run the analyses and report the results efficiently while maintaining the results' reproducible nature. The framework implements numerous features allowing state-of-the-art deep learning-based analysis. At the same time, the AISY framework allows easy add-ons of user-custom functionalities.

1 Introduction

In recent years, deep learning-based side-channel analysis (SCA) became a standard for the research community. Simultaneously, the industry is also becoming more interested that such techniques become a standard in the design and certification process. Unfortunately, many research results can be challenging to reproduce and also time-consuming to be implemented.

Reproducibility must be a cornerstone of science: if an experiment is not reproducible, we cannot be sure of the paper's conclusions. While the scientific community puts ever more emphasis on reproducability [1], this is not always easy to achieve. Indeed, the level of the details reported in a publication can significantly vary (for instance, only a few years ago, it was often seen in SCA that research works do not report on precise machine learning architectures used in the experiments). Additionally, issues can arise from different user experience levels and differences in the implementation processes or frameworks used. What is more, security experts do not necessarily have a background in artificial intelligence, making various results and settings more open for a subjective interpretation.

While it is evident that the SCA community is improving in both the quality of the results and their reproducibility, there is still room for progress. In this paper, we introduce the AISY framework intended for the deep learning-based side-channel analysis. AISY offers all commonly used settings and allows users

[1] See, e.g., `https://ches.iacr.org/2021/artifacts.php`.

to extend the framework according to their needs easily. More precisely, AISY was developed to be:

1. **Easy to use**. AISY framework allows very easy execution of deep learning in profiling side-channel attacks. The framework is built on top of *Keras* library (integrated in *TensorFlow* library) and users that familiar to basic *Keras*'s functionalities can easily extent the framework.
2. **Integrated Database**. AISY framework comes with the option to store all analysis results in an SQLite database. Standard libraries are implemented in the framework, and users can easily add custom tables to the framework. The creation of custom tables does not require any specific background knowledge on databases.
3. **Web application**. AISY framework is also integrated with *Flask* python-based web framework. A web application is integrated with a web-based user interface. The web application provides a user-friendly way to visualize analysis, plots, results, and tables.
4. **One-click Script Generation**. A user can generate the full script used to produce results stored in the web application database. This is particularly important when the user wishes to reproduce results after he changed the script. Another advantage of this feature can be seen when a user shares the database with a second user. The latter generate all the scripts from the database.
5. **State-of-the-art side-channel analysis**. AISY framework brings state-of-the-art deep learning-based side-channel attacks. We follow recent publications and keep the framework updated accordingly. Version 0.1 already provides state-of-the-art features like ensembles, hyperparameter search, visualization, data augmentation, and special SCA metrics.
6. **Teamwork**. AISY framework allows easy and efficient teamwork. The structure is done so that multiple users can easily share results and contribute together to analysis. By sharing the database, users in different locations can visualize results without sending multiple emails with files, figures, and code.

Since the framework offers a simple way of generating scripts used to obtain specific results, we believe it will significantly increase future work reproducibility. AISY framework is open-source and planned to receive regular updates to offer state-of-the-art functionalities in the deep learning-based SCA.

The rest of this paper is organized as follows. Section 2 provides background theory of deep learning-based SCA. Related works are described in Section 3. Section 4 describes the AISY framework, including general design, installation aspects, and main features. In Section 5, we provide an experimental evaluation of the framework and how a user can add custom-based functionalities to the framework. Finally, conclusions and future works are discussed in Section 6.

2 Background

Let calligraphic letters like $\mathcal{X}$ denote sets, and the corresponding upper-case letters X denote random variables and random vectors $\mathbf{X}$ over $\mathcal{X}$. The corresponding lower-case letters x and $\mathbf{x}$ denote realizations of X and $\mathbf{X}$, respectively. Let k be a key candidate that takes its value from the keyspace $\mathcal{K}$, and k^* the correct key.

We define a dataset as a collection of traces (measurements) $\mathbf{T}$, where each trace $\mathbf{t}_i$ is associated with an input value (plaintext or ciphertext) $\mathbf{d}_i$ and a key $\mathbf{k}_i$. The dataset size equals S, representing the sum of the training set size N, validation set size V, and test set size Q. If we consider only a specific key byte, we denote the key byte as $k_{i,j}$, and input byte as $d_{i,j}$. Finally, $\boldsymbol{\theta}$ denotes the vector of parameters to be learned in a profiling model (e.g., the weights in neural networks), and $\mathcal{H}$ denotes the set of hyperparameters defining the profiling model.

2.1 Supervised Learning and Profiling SCA

Supervised machine (deep) learning represents the machine learning task of learning a function f that maps an input to the output ($f : \mathcal{X} \rightarrow Y)$) based on examples of input-output pairs. The function f is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^n$, where n denotes the number of trainable parameters.

While supervised learning can be used for different tasks (e.g., binary classification, multi-class classification, regression), we consider only the multi-class classification task in the rest of this paper. As such, we consider the c-classification task, where c denotes the number of classes. More precisely, the classifier is a function that maps input samples to label space ($f : \mathcal{X} \rightarrow \mathbb{R}^c$)[2].

Supervised classification happens in two phases: training and test.

1. The goal of the training phase is to learn the parameters $\boldsymbol{\theta}'$ that minimize the empirical risk represented by a loss function L on a dataset T of size N ($T = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$):

$$\boldsymbol{\theta}' = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i}^{N} L(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i). \tag{1}$$

In the rest of this paper, the function f is a deep neural network with the *Softmax* output layer. Additionally, we encode classes in one-hot encoding, where each class is represented as a vector of c values that has zero on all the places, except one place, denoting the membership of that class, i.e., $\mathbf{y}_i = \mathbf{e}_{y_i} \in \{0,1\}^c$ such that $1^T \mathbf{y}_i = 1 \; \forall i$. The common loss function for deep learning is the categorical cross-entropy:

$$CCE = -\sum_{i}^{c} \mathbf{y}_{ij} \log(f_j(\mathbf{x}_i, \boldsymbol{\theta}), \tag{2}$$

[2] (Is more common to use the notion of features than samples in SCA, but in the rest of the paper, we use sample to differentiate from the AISY framework features)

where $\mathbf{y}_{ij}$ corresponds to the j element of one-hot encoded class for input $\mathbf{x}_i$, and f_j denotes the j element of f.
To monitor the performance of the model f, it is common to evaluate the model on a separate validation set and stop the training process once there is a discrepancy between the metrics for the training and validation sets (to prevent the overfitting). Overfitting happens when a model learns the detail and noise in the training data, negatively impacting the model's performance on unseen data. Common metrics to observe are training/validation accuracy and loss.

2. In the test phase, the goal is to make predictions about classes y based on the previously unseen examples $\mathbf{x}$ (where the number of examples is Q), and the trained model f. A common metric in machine learning to evaluate a classifier's performance is accuracy (but others like precision, recall, or F1 score are often used).

When considering profiling SCA, the profiling phase is the same as the training. On the other hand, there are slight differences in SCA's attack phase compared to the testing phase in machine learning. In the attack phase, the goal is to make predictions about the classes

$$y(\mathbf{x}_1, k^*), \ldots, y(\mathbf{x}_Q, k^*),$$

where k^* represents the secret (unknown) key on the device under the attack. The outcome of predicting with a model f on the attack set is a two-dimensional matrix P with dimensions equal to $Q \times c$. Every element $\mathbf{p}_{i,v}$ of matrix P is a vector of all class probabilities for a specific trace $\mathbf{x}_i$ (note that $\sum_v^c \mathbf{p}_{i,v} = 1, \forall i$. The probability $S(k)$ for any key byte candidate k is a valid SCA distinguisher, where it is common to use the maximum log-likelihood distinguisher:

$$S(k) = \sum_{i=1}^{Q} \log(\mathbf{p}_{i,v}). \tag{3}$$

The value $\mathbf{p}_{i,v}$ denotes the probability that for a key k and input d_i, we obtain the class v. The class v is derived from the key and input through a cryptographic function CF and a leakage model l.

In SCA, an adversary is commonly not interested in predicting the classes in the attack phase but aims to reveal the secret key k^*. For this, common measures are the success rate (SR) and the guessing entropy (GE) of a side-channel attack [32]. In particular, let us assume, given Q amount of traces in the attack phase, an attack outputs a key guessing vector $\mathbf{g} = [g_1, g_2, \ldots, g_{|\mathcal{K}|}]$ in decreasing order of probability. So, g_1 is the most likely and $g_{|\mathcal{K}|}$ the least likely key candidate. Guessing entropy is the average position of k^* in $\mathbf{g}$. Commonly, averaging is done over 100 independent experiments to obtain statistically significant results. Note that while defined here as guessing entropy for the whole key, it can also be observed for separate key bytes, in which case it is called partial guessing entropy. In this work, we calculate partial guessing entropy (i.e., one key byte only), but we denote it as guessing entropy for simplicity.

3 Related Works

Template Attack

The first profiling attack is the template attack, where the authors showed it could break implementations secure against other forms of side-channel attacks [4]. This attack is the most powerful one from the information-theoretic perspective, and at the same time, it has no hyperparameters to tune. The main drawbacks of this attack are unrealistic assumptions (unlimited number of traces, noise following the Gaussian distribution [17]), which makes this attack less powerful than, e.g., deep learning-based attacks for many realistic settings. Researchers suggested using one pooled covariance matrix averaged over all labels to cope with statistical difficulties arising in template attack when the number of samples is larger than the number of traces per class [5].

Machine Learning-based SCA

From the "classical" machine learning techniques, there are extensive works considering techniques like random forest [16, 19], support vector machines [12, 13, 29], and Naive Bayes [11, 27]. Additionally, in previous years researchers often used multilayer perceptron, but either the architectures that are not deep learning [3] or settings where the sample selection is conducted to reduce the computational complexity [6, 20, 21, 42].

Supervised Learning and the Attack Performance

Considering the deep learning approaches, the last five years showed a plethora of papers using various techniques (mostly, multilayer perceptron and convolutional neural networks). As such, the next paragraphs are not intended to serve as an exhaustive list of deep learning-based SCAs. For more details on deep learning-based SCA, we refer interested readers to the survey by Hettwer et al. [9].

Maghrebi et al. made a significant step forward in the profiling SCA research as they showed that convolutional neural networks could efficiently break targets [19]. What is more, they also showed how deep learning could work with raw traces, removing the need for various sample selection techniques [26]. Cagli et al. showed how deep learning breaks implementations protected with countermeasures [3]. Additionally, they introduced the concept of data augmentation in the profiling SCA. Picek et al. evaluated a number of machine learning metrics showing a discrepancy from the side-channel metrics [28]. Additionally, the authors showed that the metrics issue happens not only for "classical" machine learning techniques but also deep learning. Kim et al. took inspiration from convolutional neural networks developed for the speech domain and showed top results on various SCA datasets [14]. To further improve the attack performance,

[3] Or, at least, not clear if deep learning architectures since the number of hidden layers is not reported.

they showed how Gaussian noise could be added to the input to serve as a regularization factor. Hettwer et al. introduced the concept of domain knowledge to the deep learning-based SCA [8]. More precisely, the authors encode additional information (plaintext or ciphertext) as an additional input to the neural network. Perin et al. investigated the performance of a random search-based hyperparameter tuning to build ensembles of deep learning-based classifiers [23]. Bhasin et al. investigated the deep learning performance when considering the differences between the training and test device (different key and/or different device) [2]. Their results indicate that deep learning can easily overfit, so proper training is of utmost importance. The authors also suggested the MDM model where multiple devices are used for training/validation.

Hyperparameter Tuning

Benadjila et al. introduced the ASCAD database for the experimentation with profiling attacks [1]. Today, this dataset represents a common choice for evaluating deep learning-based attacks. Additionally, the authors provided an investigation on the importance of hyperparameter tuning. Zaid et al. proposed a methodology to select hyperparameters related to the size (number of learnable parameters, i.e., weights and biases) of layers in CNNs [43]. The hyperparameters include the number of filters, kernel sizes, strides, and neurons in fully-connected layers. Wouters et al. [38] improved upon the work from Zaid et al. [43] and discussed several misconceptions in the first work. Additionally, they showed how to reach similar attack performance with significantly smaller neural network architectures.

L. Weissbart investigated hyperparameter tuning for multilayer perceptrons and showed that MLP with proper hyperparameter tuning could break implementations with different countermeasures [37]. Li et al. investigated the weight initialization role for MLP and CNN architectures and showed that depending on the choice of the weight initialization method, SCA attack performance could vary significantly [18]. Perin and Picek explored the influence of the optimizer choice for deep learning-based side-channel analysis [24]. Their results indicated that commonly used optimizers easily overfit, requiring more care during the training process. Van der Valk et al. considered a setup where a small neural network is learned to mimic the behavior of a larger neural network, showing that small neural networks can indeed reach top attack performance [36]. Perin et al. showed how pruning could help to obtain significantly smaller neural networks that outperform larger neural networks [25]. Additionally, they demonstrated how pruning a neural network helps break the targets even when larger neural networks cannot break them.

Wu et al. introduced Bayesian optimization for hyperparameter tuning [39]. There, the authors managed to find small neural network architectures that perform well (surpassing the architectures' performance obtained by the methodologies mentioned above). Rijsdijk et al. investigated how reinforcement learning can be used to find small convolutional neural networks that perform well, surpassing the previous state-of-the-art [31]. Knezevic et al. used evolutionary algorithms to evolve custom activation functions for SCA [15]. The authors showed that activation functions significantly different from those commonly used could bring excellent attack performance.

Unsupervised Paradigm

Wu and Picek used denoising autoencoders to remove the influence of side-channel countermeasures from the measurements [40]. Ramezanpour et al. used autoencoders to conduct unsupervised sample selection [30]. B. Timon considered non-profiling deep learning SCA where an attacker can only collect a limited number of side-channel traces for a fixed unknown key value from a closed device [33].

Toward Explainable Deep Learning-based SCA

Hettwer et al. considered the selection of points of interest for deep learning by using three deep neural network attribution methods [10]. Masure et al. used gradient visualization to discover where the sensitive information leaks [22]. Van der Valk and Picek extended guessing entropy to guessing entropy bias-variance decomposition to improve the interpretability of results from machine learning-based attacks [34]. Van der Valk et al. explored the information obtained from the activation functions in neural networks to interpret what neural networks learn while training on different SCA datasets [35]. Wu et al. investigated the differences in the deep learning-based SCA stemming from the different guessing entropy estimations [41].

4 AISY Framework

4.1 General Design

The AISY framework is a (Keras/TensorFlow) deep learning-based framework for profiling side-channel analysis. This project was implemented as a result of several years of research on deep learning and side-channel analysis by AISyLab at TU Delft, The Netherlands. The current framework version is 1.0.

The project is open-source and available at `https://github.com/AISyLab/AISY_Framework`. The documentation about the project is available at `https://aisylab.github.io/AISY_docs/`. Currently, the AISY framework supports deep learning-based SCA for the AES cipher with 128-bit key [4].

In Figure 1, we depict the AISY framework flow. The figure illustrates the main framework flow. In blue color, we depict the basic operations of the framework (those that always execute). In light orange color, we depict the optional features. Note that the AISY framework provides a web application to visualize results when stored in the database files.

[4] New releases will support several cryptographic algorithms.

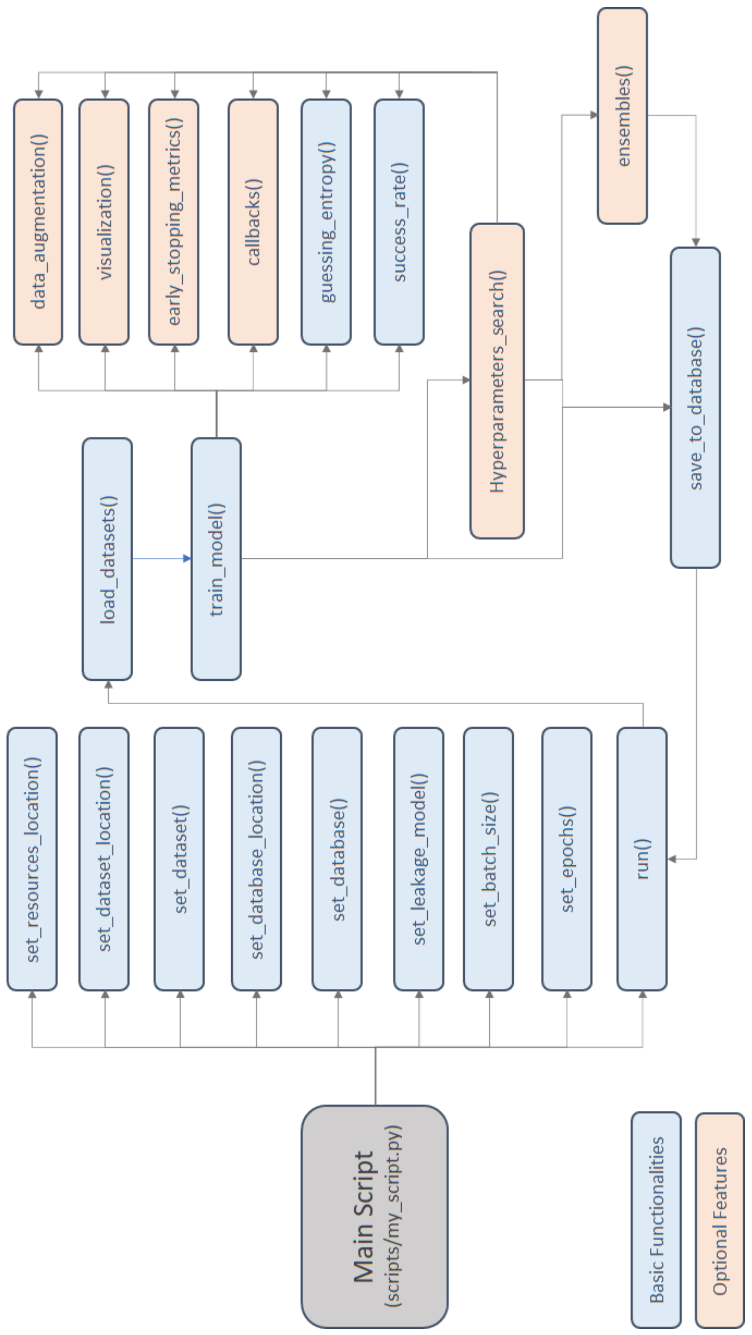


Fig. 1: The AISY framework flow. In blue color, we depict the basic operations of the framework (those that always execute). In light orange color, we depict the optional features.

4.2 Installation

The AISY framework can be installed from the AISyLab github page:

```
git clone https://github.com/AISyLab/AISY_Framework.git
cd AISY_framework
pip install -r requirements.txt
```

Finally, to open the web application, one needs to run the following command:

```
flask run
```

The framework has the following layout:

```
custom/                              # folder containing customized definitions
    custom_callbacks/callbacks.py     # file with user callbacks
    custom_metrics/                   # each .py file contain a custom metric
    custom_models/neural_networks.py  # file to insert user neural networks (keras models)
    custom_datasets/datasets.py       # file with dataset details
    custom_data_augmentation/         # file containing data augmentation method
    custom_tables/tables.py           # file containing custom sqlite tables
resources/                           # folder to store user resources (created when first analysis executed)
    databases/                        # .sqlite database files with project information and analysis results
    figures/                          # .png figure generated from user
    models/                           # .h5 models
    npz/                              # .npz files with project information and analysis results
scripts/                             # folder to store main user scripts
webapp/                              # flask web application files (html, js, css)
app.py                               # main flask application
```

4.3 Datasets

Currently, there are five datasets supported in the AISY framework:

- **ASCAD Fixed Key**. The first target platform is an 8-bit AVR microcontroller running a masked AES-128 implementation, where the side-channel is electromagnetic emanation [1]. This dataset consists of 50 000 traces for profiling and 10 000 for testing. Commonly, researchers attack the first masked byte, which is key byte three, and for it, the authors provided the pre-selected window of 700 samples. This dataset is available at `https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1/ATM_AES_v1_fixed_key`.
- **ASCAD Random Keys**. This dataset is obtained from the same target as the ASCAD Fixed Key dataset. This dataset has random keys in the profiling set, which has 200 000 traces, and a fixed key in the test set that has 100 000 traces. The pre-selected window to attack the first masked key byte (key byte three) has 1 400 samples. This dataset is available at `https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1/ATM_AES_v1_variable_key`.
- **CHES CTF 2018**. The traces consist of masked AES-128 encryption running on a 32-bit STM microcontroller. There are 45 000 traces for the training set, which contains a fixed key. The attack set consists of 5 000 traces. The key used in the training and validation set is different from the key configured for the test set. Each trace consists of 2 200 samples. This dataset is available at `https://chesctf.riscure.com/2018/news`.

- **AES_HD**. AES_HD is an unprotected hardware implementation of AES-128 implemented on Xilinx Virtex-5 FPGA of a SASEBO GII evaluation board. This dataset contains 50 000 traces corresponding to 50 000 random plaintexts where each trace has 1 250 samples. This dataset is available at `https://github.com/AISyLab/AES_HD`.
- **AES_HD_ext**. The final dataset is the AES_HD extended dataset that contains 500 000 traces (corresponding to 500 000 random plaintexts) where each trace has 1 250 samples. This dataset is available at `https://github.com/AISyLab/AES_HD_Ext`.

The only format currently supported is .h5, where datasets need to be generated according to the ASCAD database description [1]. Note that the datasets are not included in the framework but can be easily downloaded and used with the framework due to compatible data formats. In Appendix A, we provide a code example of how to generate .h5 datasets using code provided by the ASCAD codebase repository.

4.4 Databases

AISY framework makes use of SQLite database to store the results of the side-channel analysis. In the main script, the user enters the name of the SQLite database file. If the file already exists, then a new entry is stored for the deployed analysis. The database is based on the SQLAlchemy Python package.

4.5 Standard Metrics

AISY framework provides four standard metrics: guessing entropy, success rate, accuracy, and loss. To compute guessing entropy, a user must define the key rank calculation definition, as in the example below:

```
import aisy_sca
from app import *
from custom.custom_models.neural_networks import *

aisy = aisy_sca.Aisy()
aisy.set_resources_root_folder(resources_root_folder)
aisy.set_database_root_folder(databases_root_folder)
aisy.set_datasets_root_folder(datasets_root_folder)
aisy.set_database_name("database_ascad.sqlite")
aisy.set_dataset(datasets_dict["ascad-variable.h5"])
aisy.set_aes_leakage_model(leakage_model="HW", byte=2)
aisy.set_batch_size(400)
aisy.set_epochs(20)
aisy.set_neural_network(mlp)
aisy.set_number_of_attack_traces(20000)
aisy.run(
    key_rank_executions=100,
    key_rank_report_interval=10,
    key_rank_attack_traces=1000
)
```

Inside *run* method, we define the parameters for the guessing entropy calculation: the number of key rank executions, the number of attack traces, and the trace report (plotting) interval. The *key_rank_attack_traces* defines the number of attack traces used to compute each key rank execution. These attack traces are randomly selected out of 20 000 attack traces for each key rank execution (this is an important procedure in guessing entropy calculation for the generalization estimation of the trained model, as explained in a recent research paper [41]). Together with guessing entropy, the framework also automatically computes the success rate. Accuracy and loss are estimated for each epoch during training.

4.6 Neural Network Models

Currently, the AISY framework allows deep learning analysis with multilayer perceptron and convolution neural networks. The user can add any neural network configuration for one of these two neural network types. Below, we describe these two neural network types and give simple examples for each of them. Note that the architectures are implemented in Keras (i.e., one does not need to use any special code in the AISY framework, making the framework easy to extend).

Multilayer Perceptron The multilayer perceptron (MLP) is a feed-forward neural network that maps sets of inputs onto sets of appropriate outputs. MLP consists of multiple layers (at least three) of nodes in a directed graph, where each layer is fully connected to the next one, and training of the network is done with the backpropagation algorithm [7].

The listing below shows an MLP architecture consisting of four hidden layers where each hidden layer has 200 neurons and the *selu* activation function. Additionally, we use the Adam optimizer with a 0.001 learning rate and categorical cross-entropy as the loss function.

```
from tensorflow.keras.optimizers import *
from tensorflow.keras.layers import *
from tensorflow.keras.models import *

def mlp(classes, number_of_samples):
    model = Sequential()
    model.add(Dense(200, activation='selu',
    input_shape=(number_of_samples,)))
    model.add(Dense(200, activation='selu'))
    model.add(Dense(200, activation='selu'))
    model.add(Dense(200, activation='selu'))
    model.add(Dense(classes, activation='softmax'))
    model.summary()
    optimizer = Adam(lr=0.001)
    model.compile(loss='categorical_crossentropy',
    optimizer=optimizer, metrics=['accuracy'])
    return model
```

Convolutional Neural Networks Convolutional neural networks (CNNs) commonly consist of three types of layers: convolutional layers, pooling layers, and fully-connected layers. The convolution layer computes the output of neurons connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. Pooling decrease the number of extracted samples by performing a down-sampling operation along the spatial dimensions. The fully-connected layer (the same as in MLP) computes either the hidden activations or the class scores.

In the listing below, we depict a definition of a CNN consisting of one convolution layer and two fully-connected layers. All the layers use the *ReLU* activation function, and the fully-connected layers have 128 neurons each. As in the previous example, we use the Adam optimizer with a 0.001 learning rate and categorical cross-entropy as the loss function.

```
from tensorflow.keras.optimizers import *
from tensorflow.keras.layers import *
from tensorflow.keras.models import *

def cnn(classes, number_of_samples):
    model = Sequential()
    model.add(Conv1D(filters=8, kernel_size=20, strides=1,
    activation='relu', padding='valid', input_shape=(number_of_samples, 1)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu',
    kernel_initializer='random_uniform', bias_initializer='zeros'))
    model.add(Dense(128, activation='relu',
    kernel_initializer='random_uniform', bias_initializer='zeros'))
    model.add(Dense(classes, activation='softmax'))
    model.summary()
    optimizer = Adam(lr=0.001)
    model.compile(loss='categorical_crossentropy',
    optimizer=optimizer, metrics=['accuracy'])
    return model
```

To allow easier usage of the AISY framework, we also implemented several state-of-the-art architectures (the models are defined in *custom/custom_models/neural_networks.py*):

1. ASCAD_mlp [1]
2. ASCAD_cnn [1]
3. methodology_cnn_ascad [43]
4. methodology_cnn_aeshd [43]
5. methodology_cnn_aesrd [43]
6. methodology_cnn_dpav4 [43]

4.7 Leakage Models

AISY framework supports four different leakage models:

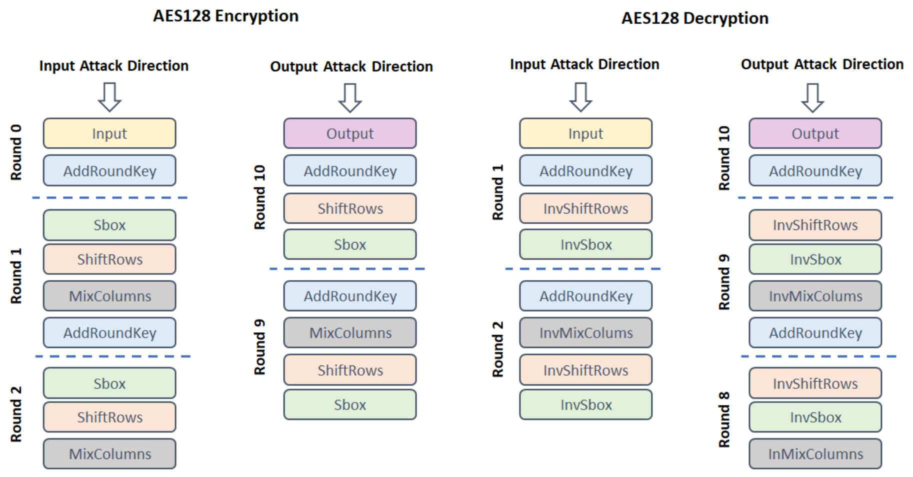


Fig. 2: Supported AES states for leakage model definitions.

1. **Bit**. This leakage model results in two classes as every bit can be either value 0 or value 1.
2. **Hamming weight (HW)**. This leakage model results in nine classes for the AES key byte (from the Hamming weight 0 to the Hamming weight 8).
3. **Hamming distance (HW)**. This leakage model results in nine classes for the AES key byte (from the Hamming distance 0 to the Hamming distance 8). Differing from the Hamming weight leakage model, we need to consider two states that are XOR-ed to obtain the intermediate value.
4. **Identity (ID)**. This leakage model considers the value of the intermediate state. For one key byte, it results in 256 possible classes.

These leakage models can be applied to any AES round key byte (or bit in byte) in both encryption and decryption modes. Additionally, we support the following target states:

– **Input**. The input value of the cipher round.
– **Sbox/InvSbox**. State after the S-box or inverse S-box operation.
– **ShiftRows/InvShiftRows**. State after the ShiftRows or Inverse ShiftRows operation.
– **MixColumns/InvMixColumns**. State after the MixColumns or Inverse MixColumns operation.
– **AddRoundKey**. State after XOR operation with the round key.
– **Output**. The output value of the cipher round.

Figure 2 illustrates all the possible target AES states in encryption and decryption executions.

4.8 Visualization

AISY provides an input gradient visualization feature. This feature allows the visual verification of main input samples learned from the input traces. For example, we provide the following listing code that would visualize the input gradients on 4 000 training traces from the ASCAD random keys dataset. Note that the input gradient can be visualized as:

1. the sum of input gradients, providing the sum of input gradients computed for all used profiling traces and all the processed epochs,
2. the input gradient computed for all used profiling traces for each epoch in a heatmap plot.

```
import aisy_sca
from app import *
from custom.custom_models.neural_networks import *

aisy = aisy_sca.Aisy()
aisy.set_resources_root_folder(resources_root_folder)
aisy.set_database_root_folder(databases_root_folder)
aisy.set_datasets_root_folder(datasets_root_folder)
aisy.set_database_name("database_ascad.sqlite")
aisy.set_dataset(datasets_dict["ascad-variable.h5"])
aisy.set_aes_leakage_model(leakage_model="HW", byte=2)
aisy.set_batch_size(400)
aisy.set_epochs(20)
aisy.set_neural_network(mlp)

aisy.run(visualization=[4000])
```

The above code produces visualization results (input gradients) similar to the results shown in Figure 3 (results obtained from the web application).

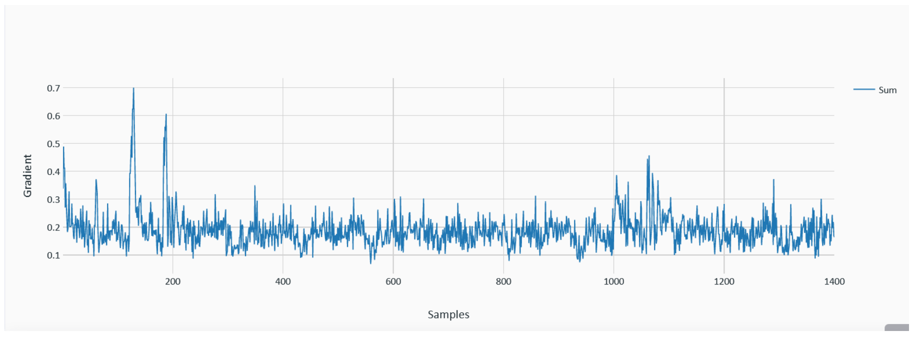
4.9 Data Augmentation

AISY framework allows easy configuration of data augmentation techniques during model training. Data augmentation is a common machine learning technique to improve model learnability. Data augmentation in the AISY framework allows small modifications in side-channel traces during training, which improves the model generalization. Currently, AISY implements two data augmentation techniques:

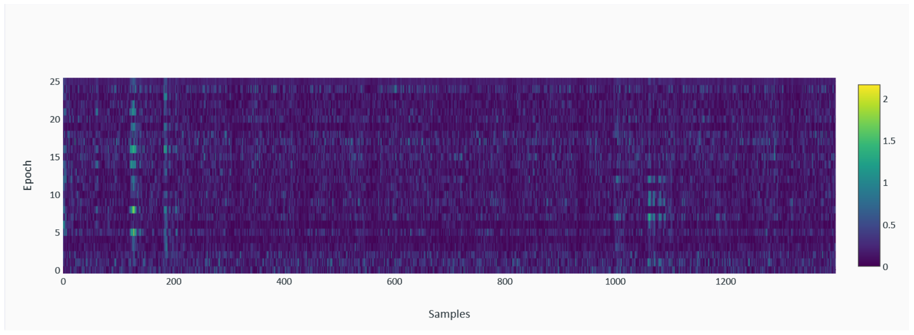
1. Shifts - every trace is randomly shifted (thus, this changes a trace in the temporal domain).
2. Gaussian noise - every trace is combined with the Gaussian noise with a specif mean and standard deviation values (thus, this changes a trace in the magnitude).

Note that the user can easily change the magnitude of added perturbations, as seen in the example below for the Gaussian noise where the user adjusts the mean and standard deviation values:

```
    noise = np.random.normal(0, 1, ns)
```



(a) Sum of input gradients for all processed epochs.



(b) Heatmap of Input gradients for each processed epoch.

Fig. 3: Visualization results.

4.10 Hyperparameter Search

Hyperparameters are configuration variables external to the model f, e.g., the number of hidden layers in a neural network. There are two options to conduct hyperparameter tuning in the AISY framework. The first option is to use the random search, where we need to define the minimal, maximal, and step value for every hyperparameter. For instance,

```
'neurons': {"min": 10, "max": 1000, "step": 10},
```

defines that the number of neurons in a layer can be any number between 10 and 1 000 in steps of 10.

The second option for hyperparameter tuning is the grid search. There, one defines all hyperparameter values to examine. Note that the grid search results in the number of settings equal to the Cartesian product of all hyperparameter dimensions.

4.11 Automatically Generated (Reproducible) Scripts

One of the main features of the AISY framework is one-click script generation. From the list of the stored database records (analyses), one can automatically generate the script used to produce the results. The advantage of this feature is the possibility to reproduce results from older analysis when the script is updated or even modified by accident. Another advantage is that if databases are shared between the users, this functionality allows different users to reproduce the same analysis.

In the web-based application interface, the user has access to buttons to generate python code used to produce the specific results (e.g., those given in Section 5.1). The user has two options:

1. Generate script: this option generates the code structure to repeat the experiment displayed on the web-application page. The generated script cannot provide a fully reproducible result (seeds and other random variables are not provided with the script). However, it is interesting for other types of investigations (e.g., analysis consistency).
2. Generate reproducible script: this option generates fully reproducible scripts, including seeds and random numbers generated from the original analysis and stored in the databaseThese values are created for any method used for weight initialization in neural networks and guessing entropy (resp. success rate) calculations. With this option, a user can generate identical results from an analysis stored in the database.

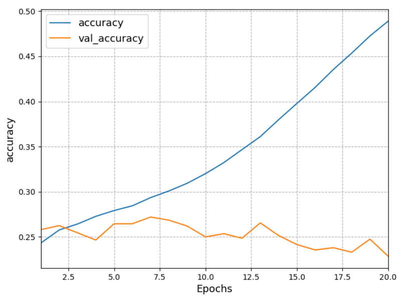
5 Experimental Evaluation

In this section, we first start by providing an example of the results obtained with our framework. Afterward, we discuss how difficult it is to extend the framework with new functionalities.

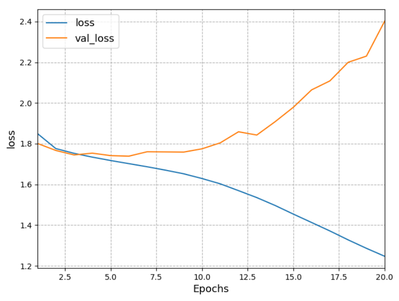
5.1 ASCAD Random Keys Example

We provide an example of the AISY framework results obtained for the ASCAD random keys dataset. As detailed in the code below, we consider the HW leakage model and attack key byte 3. The experiment uses a multilayer perceptron architecture (see Section 4.6), runs for 20 epochs, and has a batch size of 400.

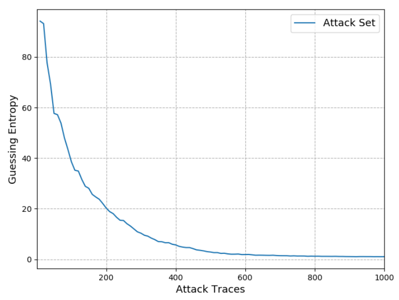
```python
import aisy_sca
from app import *
from custom.custom_models.neural_networks import *

aisy = aisy_sca.Aisy()
aisy.set_resources_root_folder(resources_root_folder)
aisy.set_database_root_folder(databases_root_folder)
aisy.set_datasets_root_folder(datasets_root_folder)
aisy.set_database_name("database_ascad.sqlite")
aisy.set_dataset(datasets_dict["ascad-variable.h5"])
aisy.set_aes_leakage_model(leakage_model="HW", byte=2)
aisy.set_batch_size(400)
```



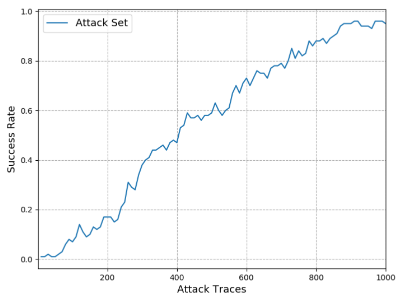
(a) Accuracy vs Epochs.



(b) Loss vs Epochs.



(c) Guessing entropy.



(d) Success Rate.

Fig. 4: Results on the ASCAD random keys dataset.

```
aisy.set_epochs(20)
aisy.set_neural_network(mlp)
aisy.run()
```

By running the above script, the user will obtain results similar to the results displayed in Figure 4. Here, we depict the results with standard metrics as discussed in Section 4.5: accuracy, loss, guessing entropy, and success rate.

5.2 Functionality Upgrades

AISY framework allows easy add-ons, including neural network models, datasets, metrics, database tables, and (Keras) callbacks. To add any of these functionalities, the user needs to follow basic guidelines, as explained below.

Adding a Neural Network Architecture. As detailed in the last section, the basic structure of the framework contains a *custom* folder with a *custom_models* sub-folder. This sub-folder contains a *neural_networks.py* file with the description of custom neural network models. The user can freely rename this file or create new python files with neural network descriptions. The only modification in the main script is the import line containing the model definition:

```
from custom.custom_models.neural_networks import *
```

Adding a Dataset. In Section 4.3, we provide a description of datasets that are already supported by the AISY framework. The concept of "support" here means that *custom/custom_datasets/datasets.py* file already contains the definitions of these datasets, including file name, (fixed) key of attack set, number of profiling traces, number of attack traces, and number of samples. In the documentation page (`https://aisylab.github.io/AISY_docs/datasets/`), we provide details on how to download these datasets to a local disk.

To add a new dataset, a user must create an .h5 format dataset (the current AISY framework version only has support for .h5 formats). The session *Creating datasets for AISY framework* in `https://aisylab.github.io/AISY_docs/datasets/` provides a code example (the code was created by ANSSI and originally released in ASCAD database repository. We made small modifications to this code in order to adapt to our needs.

As indicated in `https://aisylab.github.io/AISY_docs/datasets/`, the user needs to provide the following arrays to generate a custom dataset:

– ***train_samples***: 2d-array containing train or profiling trace samples. Every row in this array is a trace and the number of columns is the trace length (number of samples).
– ***test_samples***: : 2d-array containing test or attack trace samples. Every row in this array is a trace and the number of columns is the trace length (number of samples).
– ***profiling_plaintext***: if the side-channel attack is implemented from the input of the target algorithm, the user needs to provide plaintext data. The *profiling_plaintext* is a 2d-array where each row contains plaintext bytes.
– ***profiling_ciphertext***: if the side-channel attack is implemented from the output of the target algorithm, the user needs to provide ciphertext data. The *profiling_ciphertext* is a 2d-array where each row contains cipheretxt bytes.
– ***profiling_key***: this is 2d-array where each row contains the key bytes for each profiling trace.
– ***attack_plaintext***: 2d-array containing the plaintext bytes for the attacking dataset.
– ***attack_ciphertext***: 2d-array containing the ciphertext bytes for the attacking dataset.
– ***attack_key***: his is 2d-array where each row contains the key bytes for each attacking trace.

Note that the example code has no support to process *mask* data from protected targets (as provided by the ASCAD database). The user can add masks to .h5 dataset files. However, the main framework class (AISY) has no support to interpret dataset masks. After the .h5 dataset is created, the user can simply add the definitions to the dictionary located in *custom/custom_datasets/datasets.py* file, as in the example below for the ASCAD random keys dataset:

```
datasets_dict = {
    "ascad-variable.h5": {
        "filename": "ascad-variable.h5",
        "key": "00112233445566778899AABBCCDDEEFF",
        "first_sample": 0,
        "number_of_samples": 1400,
        "number_of_profiling_traces": 100000,
        "number_of_attack_traces": 1000
    }
}
```

In Appendix A, we provide a code example of how to generate .h5 datasets using the code provided by the ASCAD codebase repository.

Adding a Metric. In Section 4.5, we describe the standard metrics that are already implemented in the AISY framework: guessing entropy, success rate, accuracy, and loss. Although these are the most commonly used metrics in deep learning-based SCA, the research community is often proposing new interesting metrics to evaluate deep learning models in the context of profiling side-channel analysis [22, 41]. Therefore, our framework allows easy customization of metrics to be evaluated during neural network training.

The custom metric is immediately evaluated as an early-stopping metric. This means that for each epoch, the metric is computed, and, at the end of the training, guessing entropy and success rate are computed for the model predictions associated with the best epoch according to the custom metric.

The next code snippet is an example script with a custom metric to evaluate the best epoch according to the minimal number of attack traces to reach guessing entropy equal to 1:

```
import aisy_sca
from app import *
from custom.custom_models.neural_networks import *

aisy = aisy_sca.Aisy()
aisy.set_resources_root_folder(resources_root_folder)
aisy.set_database_root_folder(databases_root_folder)
aisy.set_datasets_root_folder(datasets_root_folder)
aisy.set_database_name("database_ascad.sqlite")
aisy.set_dataset(datasets_dict["ascad-variable.h5"])
aisy.set_aes_leakage_model(leakage_model="HW", byte=2)
aisy.set_batch_size(400)
aisy.set_epochs(10)
aisy.set_neural_network(mlp)

early_stopping = {
    "metrics": {
        "number_of_traces": {
            "direction": "min",
            "class": "custom.custom_metrics.number_of_traces",
            "parameters": []
        }
    }
}

aisy.run(
    early_stopping=early_stopping,
    key_rank_attack_traces=500
)
```

Note from the code above that the user needs to specify the class location of custom metrics. For better code organization, we recommend to place the custom metric file (e.g., *number_of_traces.py*) in *custom/custom_metrics* folder. In `https://aisylab.github.io/AISY_docs/custom_metrics`, we provide details on defining the custom metric .py file.

Adding a Callback. Callbacks are classes from the Keras/TensorFlow library containing methods to evaluate the trained model at the end of the batch, epoch, or training processes. As different deep learning applications to profiling side-channel analysis may require the model analysis from different perspectives (e.g., a user wants to analyze the magnitude of specific weights for each epoch), the AISY framework provides an easy way to add user-custom callbacks to the trained model.

The code below provides an example of how to create a custom callback class for the AISY framework, named as *CustomCallback1*:

```
class CustomCallback1(Callback):
    def __init__(self,
                 x_profiling, y_profiling, plaintexts_profiling,
                 ciphertext_profiling, key_profiling,
                 x_validation, y_validation, plaintexts_validation,
                 ciphertext_validation, key_validaton,
                 x_attack, y_attack, plaintexts_attack,
                 ciphertext_attack, key_attack,
                 param, leakage_model, key_rank_executions,
                 key_rank_report_interval, key_rank_attack_traces, *args):
        my_args = args[0]  # this line is mandatory
        self.param1 = my_args[0]
        self.param2 = my_args[1]

    def on_epoch_end(self, epoch, logs=None):
        print("Processing epoch {}".format(epoch))

    def on_train_end(self, logs=None):
        pass

    def get_param1(self):
        return self.param1

    def get_param2(self):
        return self.param2
```

Several parameters are passed to the callback, as defined in the *__init__* function from the example above. These parameters are

- profiling/attack/validation traces (*x_profiling*, *x_attack*, *x_validation*);
- profiling/attack/validation categorical labels (*y_profiling*, *y_attack*, *y_validation*)
- profiling/attack/validation plaintexts (*plaintexts_profiling*, *plaintexts_attack*, *plaintexts_validation*)
- profiling/attack/validation ciphertexts (*ciphertext_profiling*, *ciphertext_attack*, *ciphertext_validation*)
- profiling/attack/validation keys (*key_profiling*, *key_attack*, *key_validation*)
- target parameters: *params*;
- leakage model parameters: *leakage_model*;
- key rank parameters: *key_rank_executions*, *key_rank_report_interval*, *key_rank_attack_traces*;
- additional arguments: *args*.

In the code example above, the defined *CustomCallback1* also parses two additional parameters (*param1* and *param2*). The code below provides an example of how to execute the custom callback in the training process with these additional parameters (note the "parameters" item in the *custom_callbacks* dictionary):

```
import aisy_sca
from app import *
from custom.custom_models.neural_networks import *

aisy = aisy_sca.Aisy()
aisy.set_resources_root_folder(resources_root_folder)
aisy.set_database_root_folder(databases_root_folder)
aisy.set_datasets_root_folder(datasets_root_folder)
aisy.set_database_name("database_ascad.sqlite")
aisy.set_dataset(datasets_dict["ascad-variable.h5"])
aisy.set_aes_leakage_model(leakage_model="HW", byte=2)
aisy.set_batch_size(400)
aisy.set_epochs(10)
aisy.set_neural_network(mlp)

param1 = [1, 2, 3]
param2 = "my_string"

custom_callbacks = [
    {
        "class": "custom.custom_callbacks.callbacks.CustomCallback1",
        "name": "CustomCallback1",
        "parameters": [param1, param2]
    }
]

aisy.run(
    callbacks=custom_callbacks
)

custom_callbacks = aisy.get_custom_callbacks()

custom_callback1 = custom_callbacks["CustomCallback1"]
print(custom_callback1.get_param1())
print(custom_callback1.get_param2())
```

As specified in the above code example, the custom callback class is defined in *custom/custom_callbacks/callbacks.py* python file.

Adding a Table to Database. AISY framework uses a database python package, *aisy_database*, that implements SQLite-based database. The implementation uses the SQLAlchemy python package for the database. The *aisy_database* package contains standard database tables for basic functionalities in the AISY framework. We provide an example for the user to extend the database by creating custom tables. The AISY documentation page provides code examples to create custom tables for the AISY framework.

The code example below shows how to create a custom table easily and how to insert data into it:

```
import aisy_sca
from app import *
from custom.custom_models.neural_networks import *
from custom.custom_tables.tables import *

aisy = aisy_sca.Aisy()
aisy.set_resources_root_folder(resources_root_folder)
aisy.set_database_root_folder(databases_root_folder)
aisy.set_datasets_root_folder(datasets_root_folder)
aisy.set_database_name("database_ascad.sqlite")
aisy.set_dataset(datasets_dict["ascad-variable.h5"])
aisy.set_aes_leakage_model(leakage_model="HW", byte=2)
aisy.set_batch_size(400)
aisy.set_epochs(20)
aisy.set_neural_network(mlp)

aisy.run()

start_custom_tables(databases_root_folder + "database_ascad.sqlite")
session = start_custom_tables_session(databases_root_folder +
"database_ascad.sqlite")

new_insert = CustomTable(
    value1=10, value2=20, value3=30,
    analysis_id=aisy.get_analysis_id())
session.add(new_insert)
session.commit()
```

6 Conclusions and Future Work

This paper presents the main design principles and functionalities for the AISY framework built to support deep learning-based side-channel analysis. To the best of our knowledge, there are no other frameworks for deep learning-based side-channel analysis, and those that also include deep learning options are not publicly available (see, e.g., the Riscure Inspector tool [5]). What is more, the AISY framework is designed to be easy to use and easy to extend. We plan to update the AISY framework with state-of-the-art functionalities regularly. Through the AISY framework website, it is also possible to request new functionalities to be added.

References

1. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ASCAD database. J. Cryptographic Engineering **10**(2), 163–188 (2020). https://doi.org/10.1007/s13389-019-00220-8, `https://doi.org/10.1007/s13389-019-00220-8`
2. Bhasin, S., Chattopadhyay, A., Heuser, A., Jap, D., Picek, S., Shrivastwa, R.R.: 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), `https://www.ndss-symposium.org/ndss2020/`
3. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2017. pp. 45–68. Springer International Publishing, Cham (2017)
4. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: Jr., B.S.K., Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2523, pp. 13–28. Springer (2002). https://doi.org/10.1007/3-540-36400-5_3, `https://doi.org/10.1007/3-540-36400-5\_3`
5. Choudary, O., Kuhn, M.G.: Efficient template attacks. In: Francillon, A., Rohatgi, P. (eds.) Smart Card Research and Advanced Applications. pp. 253–270. Springer International Publishing, Cham (2014)

[5] `https://www.riscure.com/security-tools/inspector-sca`

6. Gilmore, R., Hanley, N., O'Neill, M.: Neural network based attack on a masked implementation of AES. In: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 106–111 (May 2015). https://doi.org/10.1109/HST.2015.7140247
7. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), `http://www.deeplearningbook.org`
8. Hettwer, B., Gehrer, S., Güneysu, T.: Profiled power analysis attacks using convolutional neural networks with domain knowledge. In: Cid, C., Jr., M.J.J. (eds.) Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11349, pp. 479–498. Springer (2018). https://doi.org/10.1007/978-3-030-10970-7_22, `https://doi.org/10.1007/978-3-030-10970-7\_22`
9. Hettwer, B., Gehrer, S., Güneysu, T.: Applications of machine learning techniques in side-channel attacks: a survey. J. Cryptogr. Eng. **10**(2), 135–162 (2020). https://doi.org/10.1007/s13389-019-00212-8, `https://doi.org/10.1007/s13389-019-00212-8`
10. Hettwer, B., Gehrer, S., Güneysu, T.: Deep neural network attribution methods for leakage analysis and symmetric key recovery. In: Paterson, K.G., Stebila, D. (eds.) Selected Areas in Cryptography – SAC 2019. pp. 645–666. Springer International Publishing, Cham (2020)
11. Heuser, A., Picek, S., Guilley, S., Mentens, N.: Side-channel analysis of lightweight ciphers: Does lightweight equal easy? In: Hancke, G.P., Markantonakis, K. (eds.) Radio Frequency Identification and IoT Security - 12th International Workshop, RFIDSec 2016, Hong Kong, China, November 30 - December 2, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10155, pp. 91–104. Springer (2016). https://doi.org/10.1007/978-3-319-62024-4_7, `https://doi.org/10.1007/978-3-319-62024-4\_7`
12. Heuser, A., Zohner, M.: Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines. In: Schindler, W., Huss, S.A. (eds.) COSADE. LNCS, vol. 7275, pp. 249–264. Springer (2012)
13. Hospodar, G., Gierlichs, B., Mulder, E.D., Verbauwhede, I., Vandewalle, J.: Machine learning in side-channel analysis: a first study. J. Cryptogr. Eng. **1**(4), 293–302 (2011). https://doi.org/10.1007/s13389-011-0023-x, `https://doi.org/10.1007/s13389-011-0023-x`
14. Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 148–179 (2019)
15. Knezevic, K., Fulir, J., Jakobovic, D., Picek, S.: Neurosca: Evolving activation functions for side-channel analysis. Cryptology ePrint Archive, Report 2021/249 (2021), `https://eprint.iacr.org/2021/249`
16. Lerman, L., Medeiros, S.F., Bontempi, G., Markowitch, O.: A Machine Learning Approach Against a Masked AES. In: CARDIS. Lecture Notes in Computer Science, Springer (November 2013), berlin, Germany
17. Lerman, L., Poussier, R., Bontempi, G., Markowitch, O., Standaert, F.X.: Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In: International Workshop on Constructive Side-Channel Analysis and Secure Design. pp. 20–33. Springer (2015)

18. Li, H., Krček, M., Perin, G.: A comparison of weight initializers in deep learning-based side-channel analysis. In: Zhou, J., Conti, M., Ahmed, C.M., Au, M.H., Batina, L., Li, Z., Lin, J., Losiouk, E., Luo, B., Majumdar, S., Meng, W., Ochoa, M., Picek, S., Portokalidis, G., Wang, C., Zhang, K. (eds.) Applied Cryptography and Network Security Workshops. pp. 126–143. Springer International Publishing, Cham (2020)
19. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking cryptographic implementations using deep learning techniques. In: International Conference on Security, Privacy, and Applied Cryptography Engineering. pp. 3–26. Springer (2016)
20. Martinasek, Z., Zeman, V.: Innovative method of the power analysis. Radioengineering **22**(2) (2013)
21. Martinasek, Z., Hajny, J., Malina, L.: Optimization of power analysis using neural network. In: Francillon, A., Rohatgi, P. (eds.) Smart Card Research and Advanced Applications. pp. 94–107. Springer International Publishing, Cham (2014)
22. Masure, L., Dumas, C., Prouff, E.: Gradient visualization for general characterization in profiling attacks. In: Polian, I., Stöttinger, M. (eds.) Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11421, pp. 145–167. Springer (2019). https://doi.org/10.1007/978-3-030-16350-1_9, `https://doi.org/10.1007/978-3-030-16350-1\_9`
23. Perin, G., Chmielewski, L., Picek, S.: Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 337–364 (Aug 2020). https://doi.org/10.13154/tches.v2020.i4.337-364, `https://tches.iacr.org/index.php/TCHES/article/view/8686`
24. Perin, G., Picek, S.: On the influence of optimizers in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/977 (2020), `https://eprint.iacr.org/2020/977`
25. Perin, G., Wu, L., Picek, S.: Gambling for success: The lottery ticket hypothesis in deep learning-based sca. Cryptology ePrint Archive, Report 2021/197 (2021), `https://eprint.iacr.org/2021/197`
26. Picek, S., Heuser, A., Jovic, A., Batina, L.: A systematic evaluation of profiling through focused feature selection. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(12), 2802–2815 (2019)
27. Picek, S., Heuser, A., Guilley, S.: Template attack versus bayes classifier. J. Cryptogr. Eng. **7**(4), 343–351 (2017). https://doi.org/10.1007/s13389-017-0172-7, `https://doi.org/10.1007/s13389-017-0172-7`
28. Picek, S., Heuser, A., Jovic, A., Bhasin, S., Regazzoni, F.: The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(1), 209–237 (Nov 2018). https://doi.org/10.13154/tches.v2019.i1.209-237, `https://tches.iacr.org/index.php/TCHES/article/view/7339`
29. Picek, S., Heuser, A., Jovic, A., Ludwig, S.A., Guilley, S., Jakobovic, D., Mentens, N.: Side-channel analysis and machine learning: A practical perspective. In: 2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017. pp. 4095–4102 (2017)
30. Ramezanpour, K., Ampadu, P., Diehl, W.: Scaul: Power side-channel analysis with unsupervised learning. IEEE Transactions on Computers **69**(11), 1626–1638 (2020). https://doi.org/10.1109/TC.2020.3013196

31. Rijsdijk, J., Wu, L., Perin, G., Picek, S.: Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2021/071 (2021), `https://eprint.iacr.org/2021/071`
32. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Joux, A. (ed.) Advances in Cryptology - EUROCRYPT 2009. pp. 443–461. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
33. Timon, B.: Non-profiled deep learning-based side-channel attacks with sensitivity analysis. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(2), 107–131 (2019). https://doi.org/10.13154/tches.v2019.i2.107-131, `https://doi.org/10.13154/tches.v2019.i2.107-131`
34. van der Valk, D., Picek, S.: Bias-variance decomposition in machine learning-based side-channel analysis. Cryptology ePrint Archive, Report 2019/570 (2019), `https://eprint.iacr.org/2019/570`
35. van der Valk, D., Picek, S., Bhasin, S.: Kilroy was here: The first step towards explainability of neural networks in profiled side-channel analysis. Cryptology ePrint Archive, Report 2019/1477 (2019), `https://eprint.iacr.org/2019/1477`
36. van der Valk, D., Krcek, M., Picek, S., Bhasin, S.: Learning from a big brother - mimicking neural networks in profiled side-channel analysis. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2020). https://doi.org/10.1109/DAC18072.2020.9218520
37. Weissbart, L.: Performance analysis of multilayer perceptron in profiling side-channel analysis. In: Zhou, J., Conti, M., Ahmed, C.M., Au, M.H., Batina, L., Li, Z., Lin, J., Losiouk, E., Luo, B., Majumdar, S., Meng, W., Ochoa, M., Picek, S., Portokalidis, G., Wang, C., Zhang, K. (eds.) Applied Cryptography and Network Security Workshops. pp. 198–216. Springer International Publishing, Cham (2020)
38. Wouters, L., Arribas, V., Gierlichs, B., Preneel, B.: Revisiting a methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 147–168 (Jun 2020). https://doi.org/10.13154/tches.v2020.i3.147-168, `https://tches.iacr.org/index.php/TCHES/article/view/8586`
39. Wu, L., Perin, G., Picek, S.: I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/1293 (2020), `https://eprint.iacr.org/2020/1293`
40. Wu, L., Picek, S.: Remove some noise: On pre-processing of side-channel measurements with autoencoders. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 389–415 (Aug 2020). https://doi.org/10.13154/tches.v2020.i4.389-415, `https://tches.iacr.org/index.php/TCHES/article/view/8688`
41. Wu, L., Weissbart, L., Krček, M., Li, H., Perin, G., Batina, L., Picek, S.: On the attack evaluation and the generalization ability in profiling side-channel analysis. Cryptology ePrint Archive, Report 2020/899 (2020), `https://eprint.iacr.org/2020/899`
42. Yang, S., Zhou, Y., Liu, J., Chen, D.: Back propagation neural network based leakage characterization for practical security analysis of cryptographic implementations. In: Kim, H. (ed.) Information Security and Cryptology - ICISC 2011. pp. 169–185. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
43. Zaid, G., Bossuet, L., Habrard, A., Venelli, A.: Methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(1), 1–36 (Nov 2019).

https://doi.org/10.13154/tches.v2020.i1.1-36, https://tches.iacr.org/index.php/TCHES/article/view/8391

A Code to Generate .h5 Datasets According to the ASCAD Codebase

```
out_file = h5py.File('new_dataset.h5', 'w')

profiling_traces_group = out_file.create_group("Profiling_traces")
attack_traces_group = out_file.create_group("Attack_traces")

profiling_traces_group.create_dataset(
    name="traces", data=train_samples, dtype=train_samples.dtype
)
attack_traces_group.create_dataset(
    name="traces", data=test_samples, dtype=test_samples.dtype
)

metadata_type_profiling = np.dtype(
    [
        (
            "plaintext", profiling_plaintext.dtype,
            (len(profiling_plaintext[0]),)
        ),
        (
            "ciphertext", profiling_ciphertext.dtype,
            (len(profiling_ciphertext[0]),)
        ),
        (
            "key", profiling_key.dtype,
            (len(profiling_key[0]),)
        )
    ]
)
metadata_type_attack = np.dtype(
    [
        (
            "plaintext", attack_plaintext.dtype,
            (len(attack_plaintext[0]),)
        ),
        (
            "ciphertext", attack_ciphertext.dtype,
            (len(attack_ciphertext[0]),)
        ),
        (
            "key", attack_key.dtype,
            (len(attack_key[0]),)
        )
    ]
)

profiling_metadata = np.array(
    [(
        profiling_plaintext[n],
        profiling_ciphertext[n],
        profiling_key[n]) for n, k in
        zip(profiling_index, range(0, len(train_samples)))
    ],
    dtype=metadata_type_profiling
)
profiling_traces_group.create_dataset(
    "metadata",
    data=profiling_metadata,
    dtype=metadata_type_profiling
)

attack_metadata = np.array(
    [(
        attack_plaintext[n],
        attack_ciphertext[n],
        attack_key[n]) for n, k in
        zip(attack_index, range(0, len(test_samples)))
    ], dtype=metadata_type_attack)
attack_traces_group.create_dataset(
    "metadata",
    data=attack_metadata,
    dtype=metadata_type_attack
)

out_file.flush()
out_file.close()
```