Automated Side-Channel Attacks using Black-Box Neural Architecture Search

Pritha Gupta[1], Jan Peter Drees[2] and Eyke Hüllermeier[3]

[1] Paderborn University, Paderborn, Germany, prithag@mail.uni-paderborn.de
[2] University of Wuppertal, Wuppertal, Germany, jan.drees@uni-wuppertal.de
[3] Ludwig Maximilian University of Munich, Munich, Germany, eyke@lmu.de

Abstract. The usage of convolutional neural networks (CNNs) to break cryptographic systems through hardware side-channels has enabled fast and adaptable attacks on devices like smart cards and TPMs. Current literature proposes fixed CNN architectures designed by domain experts to break such systems, which is time-consuming and unsuitable for attacking a new system. Recently, an approach using neural architecture search (NAS), which is able to acquire a suitable architecture automatically, has been explored. These works use the secret key information in the attack dataset for optimization and only explore two different search strategies using one-dimensional CNNs. We propose a NAS approach that relies only on using the profiling dataset for optimization, making it fully black-box. Using a large-scale experimental parameter study, we explore which choices for NAS, such as 1-D or 2-D CNNs and search strategy, produce the best results on 10 state-of-the-art datasets for Hamming weight and identity leakage models. We show that applying the random search strategy on 1-D inputs results in a high success rate and retrieves the correct secret key using a single attack trace on two of the datasets. This combination matches the attack efficiency of fixed CNN architectures, outperforming them in 4 out of 10 datasets. Our experiments also point toward the need for repeated attack evaluations of machine learning-based solutions in order to avoid biased performance estimates.

Keywords: Neural Architecture Search · Parameter Study · Convolutional Neural Network · Side-Channel Attack · AES

1 Introduction

When it comes to hardware side-channels, machine learning techniques have long been the tool of choice for attackers. One of the most powerful tools, deep learning, is particularly promising, as it is capable of learning very complex mappings between the secret key used in the encryption and the observed power consumption or electromagnetic emissions. One issue with using Convolutional neural networks (CNNs) is designing an appropriate architecture for the network. A good architecture that matches the requirements of the dataset at hand can perform incredibly well, sometimes even predicting the correct key after observing a single attack trace. On the other side, a bad architecture may fail to give any useful predictions, no matter how much training data it is given. The hardware side-channel community has been struggling with this, failing to come up with good and general rules for architecture designs that are suitable for arbitrary attack datasets. There is one way around this issue, though: Instead of trying to come up with architecture design guidelines by manual architecture tweaking, why not treat the choice of architecture as just another machine learning (ML) hyperparameter that can be optimized automatically? This can be achieved with neural architecture search (NAS), which explores a pre-defined hyperparameter space of CNN architectures by repeatedly training models and determining their performance. If done correctly, this approach should be able to easily identify which architecture works well for any given dataset, performing at least as well as carefully constructed manual models.

The NAS approach was first applied to hardware side-channels recently by Wu, Perin, and Picek [WPP20] and Rijsdijk et al. [Rij+21], paving the road for automatic architecture design. These approaches use the secret key of the traces in the attack dataset to perform the search for an optimal architecture [WPP20; Rij+21]. This is only acceptable in a white-box setting, as the attack dataset can no longer be used as a test dataset to get an unbiased performance estimation. The strategy for performing the actual NAS has a large impact on the quality of the final architecture, but [WPP20] only explored Random and Bayesian search strategies while [Rij+21] proposed a search strategy based on reinforcement learning. The experimental analysis of these works was also limited in scope, as only a small number of datasets were considered. Even though the state-of-the-art CNN architectures were inspired by 2-D image classification models, ML-based side-channel attacks (SCAs) have only been applied them using 1-D inputs.

1.1 Our contributions

- We propose a NAS approach that relies only on using the profiling dataset for optimization, which makes it suitable for a black-box setting. In addition, it is set up to perform several independent attacks, which produces more reliable performance estimates.

- We expand the previous NAS experiments into a large-scale parameter study, investigating the impact of search strategy by considering four different search strategies, including Greedy and Hyperband, as well as 2-D CNNs. Our evaluation is performed on 10 publicly available reference datasets in both the Identity (ID) and Hamming weight (HW) leakage model.

- We also conduct a performance comparison between the CNN architectures obtained from our NAS approach and the state-of-the-art fixed architectures proposed by Benadjila et al. [Ben+20] and Zaid et al. [Zai+19].

1.2 Related Work

Breaking cryptographic implementations using their side-channel emissions has a long history, particularly in the intelligence community, who has been aware of such issues since the 1950s. In the scientific world, a breakthrough was achieved in 1999 with the introduction of Differential Power Analysis (DPA) by Kocher, Jaffe, and Jun [KJJ99]. This attack needs to observe a large number of cryptographic operations, e.g. en- or decryptions, while measuring the power consumption or electromagnetics (EMs) of the target device. These traces consisting of thousands of measurements over time are then matched to possible computations of the executed function using statistical methods, revealing the encryption keys.

This approach of attacking the target device directly requires thousands of observations during the attack, making it difficult to execute in real-world scenarios where only a handful of traces can be obtained. If a device sufficiently similar to the target device can be obtained, for example by buying a second copy of the device, its profile (a model of its leakage) can be created by observing a large number of cryptographic operations with known keys and plaintexts. In the attack phase, fewer measurements need to be obtained from the actual target device, which are subsequently matched up with this leakage model. This scenario lead Chari, Rao, and Rohatgi [CRR03] to the develop template attacks, where several parts of an attack trace are matched to the distributions of the template traces.

Creating such a function linking secret key inputs to output traces was soon recognized to be a possible application for ML algorithms. These models are trained on the profiling traces and predict the secret key used on the attack traces. Early ML-based SCAs were already capable of dealing with measurement noise and misalignment in the traces [Hos+11]. Soon, the ML models grew more sophisticated and the attacks became more successful, capable of breaking devices that have been explicitly hardened against side-channel attacks after observing only a handful of attack traces [Hos+11; Ler+15; PHG17; LBM15; Heu+20; GHO15; CDP17; Pic+21]

Tuning a ML model properly by choosing appropriate hyperparameters is paramount for its success, with well-tuned models outperforming template attacks [Pic+17; LBM15]. Deep learning is especially promising, as it is capable of approximating any continuous function in idealized settings where the universal approximation assumption holds [Cyb89]. The deep learning networks multilayer perceptrons (MLPs) and CNNs have proven to be very powerful for performing SCAs [Zai+19; SAS21]. However, these models have to be provided with an architecture, e.g. different types of neural layers arranged after each other, each with their own parameters [Zai+19; Wou+20]. Designing an appropriate architecture can be more of an art than a science, prompting a wave of experimentation with different architectures, both created from scratch as well as existing ones taken from image classification tasks [Ben+20]. In order to alleviate this issue, Perin, Chmielewski, and Picek [PCP20] proposed using ensembles of multiple networks and aggregating their predictions. While this approach improves the generalization properties of existing CNNs architectures for hardware side-channels, it also increases the computational cost and the number of trainable parameters of the model without addressing the underlying issue of architecture design.

This challenge to design optimal neural architectures is not unique to the hardware side-channel community. It has led to active exploration in the new area of NAS, which treats the design of neural architectures as another optimization problem for which approximate solutions can be determined automatically [Ren+21]. This idea has been picked up recently by Wu, Perin, and Picek [WPP20] and Rijsdijk et al. [Rij+21] and applied to hardware side-channels for the first time. These works apply a white-box scenarios and use the attack traces for optimizing the architecture with Random and Bayesian search strategies [WPP20] or using reinforcement learning [Rij+21]. These initial investigations show very promising results, being able to produce very capable CNN and MLP models on the ASCAD benchmark dataset created by Benadjila et al. [Ben+20] and the CHES_CTF dataset.

Another novel technique for hyperparameter optimization and neural architecture design proposed by Acharya, Ganji, and Forte [AGF22] is InfoNEAT, an algorithm that evolves several neural network architectures and their hyperparameters simultaneously. Instead of the usual approach where a single neural network needs to predict the whole key byte, it trains a separate neural network for each possible key byte value, using a one-versus-all multi-class classification approach. In combination with an architecture selection based on information-theoretic metrics this makes it uniquely suited for hardware attacks, but also means that results cannot be directly compared with more traditional approaches striving for a single architecture.

2 Background

This section formally describes the supervised learning problem and how it is used to perform the profiling for SCA. In addition, we describe different leakage models (assumptions) that an attacker exploits to perform the SCA. We briefly describe the basic structure of

CNNs, which is used for solving the supervised learning problem, and how NAS approaches can be used to perform the SCA automatically.

2.1 Supervised Learning for Profiled SCA

In an SCA, an attacker wants to determine the secret key used in a cryptographic operation, e.g. an encryption operation, running on a target device he can observe. For non-profiled attacks, the attacker is limited to observing the device without access to the private key being used, relying entirely on his observations of EM radiation or power consumption, for example [Pic+21]. In many cases, it is reasonable to assume that an attacker can also gain access to a second device matching the target device, called a "profiling" device, e.g. by obtaining an identical model [Pic+21]. This enables a profiled SCA, where the attacker can build a behavioral profile of the target device by running a large number of cryptographic operations with known secret keys on the profiling device. He thus obtains a set of N observation traces $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ in the first profiling phase. Each profiling trace is represented by a d-dimensional real-valued vector, i.e. $\boldsymbol{x}_i \in \mathbb{R}^d, \forall i \in \{1, \ldots, N\}$. In the attack phase, this profile is used to recover the secret key from the observed behavior of the target device.

Application to AES-128 In Advanced Encryption Standard (AES), the non-linear `SubBytes` method is being applied byte-wise to the inputs containing round keys derived from the full secret key. Because `SubBytes` uses an input-dependent S-box array lookup, this method is usually the target for SCAs. Another advantage of targeting this method is the independent operation on each input byte, allowing independently attacking specific round key bytes. Without loss of generality, we only consider attacking a single, specific key byte in a specific round of AES-128, as the same attack can be applied to multiple key bytes across multiple AES rounds to retrieve the full key [GJS19].

Profiling Dataset Structure The attacker records the traces $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ from the profiling device. Each of these N profiling traces corresponds to a single known secret key byte $k_i \in \mathcal{K}$ (with $\mathcal{K} = \{0, ..., 255\}$) and a known plaintext byte p_i. In case the attacker used different keys for each profiling trace, the key bytes $k_1, \ldots, k_N$ are also different in each trace, while $k_1 = k_2 = \ldots = k_N = k$ if the attacker used the same key for each profiling trace. The profiling trace is then labeled with $y_i = \phi(p_i, k_i)$ using a function ϕ. The function ϕ maps the plaintext p_i and the key k_i to a value that is assumed to relate to the deterministic part of the measured leakage $\boldsymbol{x}_i$ [Pic+18]. This mapping depends on the assumed leakage model and is usually defined using the AES S-box function $sbox()$ itself: $\phi(p_i, k_i) = sbox(p_i \oplus k_i)$. This labeling results in the profiling dataset $\mathcal{D}_{\text{profiling}} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$, which is then used by the profiling supervised learning algorithm to build a profiling model.

Supervised Learning The task of the profile is to predict the secret key value k_i that was used in the cryptographic operation observed in attack trace $\boldsymbol{x}_i$, for which the true secret key value is unknown. This can be formalized as a supervised learning task, where the learner is provided with a set of training data $\mathcal{D}_{\text{profiling}} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N} \subset \mathcal{X} \times \mathcal{Y}$ of size $N \in \mathbb{N}$, with $\mathcal{X} = \mathbb{R}^d$ the input space (in our case the measured traces) and $\mathcal{Y} = \{0, \ldots, C-1\}$ the output space (the 256 possible labels or "classes" produced by $\phi(p_i, k_i)$ as defined above). The task of the learning algorithm is to find a target function $f : \mathcal{X} \to \mathcal{Y}$ which, given any query $\boldsymbol{x} \in \mathcal{X}$ as input, predicts the corresponding output y in an accurate manner. Instead of simply predicting a single label, the most commonly used approach is to give a probability score for each candidate label. This allows the attacker to use the model on more than a single attack trace, aggregating the probabilities over multiple observations.

The function f can often be parameterized by parameters $\boldsymbol{w} \in \mathbb{R}^n$, where n is the number of trainable parameters. Typically, the target function is represented using a probabilistic scoring function $S : \mathcal{X} \to [0,1]^C$, which is also parameterized by $\boldsymbol{w}$. For a given instance $(\boldsymbol{x}_i, y_i)$, this function assigns a probability score for each label, such that $\boldsymbol{s}_i \coloneqq S_{\boldsymbol{w}}(\boldsymbol{x}_i) = (s_{i,0}, \dots, s_{i,C-1})$, where $s_{i,j} \coloneqq S_{\boldsymbol{w}}(\boldsymbol{x}_i)[j]$ corresponds to the probability score for label $j \in \mathcal{Y}$ for the given instance $\boldsymbol{x}_i$. Typically, neural networks are used to estimate the parameters $\boldsymbol{w}$ of the target function f. These networks implement a scoring-function $U : \mathcal{X} \to \mathbb{R}^C$, which assigns a real-valued score for each label, such that $U_{\hat{\boldsymbol{w}}}(\boldsymbol{x}) = \boldsymbol{u} = (u_0, \dots, u_{C-1})$, where $u_j \coloneqq \boldsymbol{u}[j]$. These scores are then transformed into (pseudo-)probabilities by means of the *softmax function*:

$$S_{\hat{\boldsymbol{w}}}(\boldsymbol{x})[j] = \frac{\exp(\boldsymbol{u}[j])}{\sum_{k=0}^{C-1} \exp(\boldsymbol{u}[k])} \,. \tag{1}$$

The aim of supervised learning is to learn a $\boldsymbol{w}^*$ with minimal expected loss:

$$\boldsymbol{w}^* \in \arg\min_{\boldsymbol{w}} \int L(S_{\boldsymbol{w}}(\boldsymbol{x}), y) \, dP(\boldsymbol{x}, y) \ , \tag{2}$$

where L is a loss function $[0,1]^C \times \mathcal{Y} \to \mathbb{R}$ and P the (unknown) data-generating process. One way to approximate $\boldsymbol{w}^*$ is to minimize the *empirical risk* on the profiling dataset $\mathcal{D}_{\text{profiling}}$:

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w} \in \mathbb{R}^n} R_{emp}(\boldsymbol{w}) \tag{3}$$

with

$$R_{emp}(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} L(S_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i) = \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{s}_i, y_i) \,. \tag{4}$$

Categorical Cross-Entropy (CCE) is often used as the loss function in SCA [Pic+21]:

$$L(\boldsymbol{s}_i, y_i) = L_{CCE}(\boldsymbol{s}_i, y_i) = -\sum_{j=0}^{C-1} [\![y_i = j]\!] \log(\boldsymbol{s}_{i,j}) \,, \tag{5}$$

where $[\![z]\!]$ is the indicator function returning 1 if condition z is true and 0 otherwise. Finally, the target function f is defined as $f_{\hat{\boldsymbol{w}}}(\boldsymbol{x}) = \arg\max_{j \in \mathcal{Y}} S_{\hat{\boldsymbol{w}}}(\boldsymbol{x})[j]$.

Attack Methodology The attacker records the attack traces $\boldsymbol{x}_1, \dots \boldsymbol{x}_{N_a}$ from the device under attack, by sending N_a plaintexts (or ciphertexts) $p_1, \dots p_{N_a}$. Each of these N_a attack traces $\boldsymbol{x}_i$ corresponds to the unknown key byte $k^* \in \mathcal{K}$ (with $\mathcal{K} = \{0, \dots, K\}, K = 255$ of the device and a known plaintext p_i. In order to perform the attack, the attacker needs to consider every possible key byte candidate $k \in \mathcal{K}$. For each instance $(\boldsymbol{x}_i, p_i)$, a label is generated for every key byte candidate $k \in \mathcal{K}$ using the same $\phi(p_i, k)$ function used during the profiling phase. The resulting labels are denoted by the vector $\boldsymbol{y}_i = (y_{i,0}, \dots y_{i,K-1})$, such that $y_{i,k} = \phi(p_i, k), \forall k \in \mathcal{K}$. The labeling results in the attack dataset $\mathcal{D}_{attack} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i), \dots, (\boldsymbol{x}_N, \boldsymbol{y}_{N_a})\}$, which is then used by the learned profiling model to acquire the secret key byte k^* of the device. To perform the attack, the learned probabilistic scoring function $S_{\hat{\boldsymbol{w}}}$ is used to acquire the scores for every possible key byte candidate $k \in \mathcal{K}$. For a given attack instance $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, the scores of every key byte candidate are denoted by the vector $\hat{\boldsymbol{s}}_i \coloneqq (S_{\hat{\boldsymbol{w}}}(\boldsymbol{x}_i)[y_{i,0}], \dots S_{\hat{\boldsymbol{w}}}(\boldsymbol{x}_i)[y_{i,K-1}]) = (\hat{s}_{i,0}, \dots, \hat{s}_{i,K-1})$, such that $\hat{s}_{i,k} \coloneqq S_{\hat{\boldsymbol{w}}}(\boldsymbol{x}_i)[y_{i,k}]$ represents the score of the key byte candidate $k \in \mathcal{K}$ [Ben+20]. Using these predicted scores, the cumulative score for each key byte candidate $k \in \mathcal{K}$ is calculated over several attack traces using the maximum log-likelihood [Ben+20; Pic+21]:

$$\boldsymbol{d}_{N_a}[k] = \log\left(\prod_{i=1}^{N_a} \hat{s}_{i,k} \right) = \sum_{i=1}^{N_a} \log(\hat{s}_{i,k}) \tag{6}$$

Using the likelihood to acquire the cumulative scores is an outlier sensitive operation, as a single low score value can completely disqualify the true key [Lom+14]. To increase robustness and reduce sensitivity toward low scores, the attack is run multiple times on shuffled traces of the attack dataset to obtain the corresponding cumulative scores $\boldsymbol{d}_{N_a}[k]$.

Guessing Entropy The guessing entropy (GE) is the number of guesses that are required by a model to predict the correct key k^* [Mas94]. It is acquired using the ranking vector, which contains the position of each key: $\boldsymbol{r}_{N_a}[\tilde{k}] = 1 + \left(\sum_{k \in \mathcal{K} \setminus \tilde{k}} [\![\boldsymbol{d}_{N_a}[k] > \boldsymbol{d}_{N_a}[\tilde{k}]]\!]\right), \forall \tilde{k} \in \mathcal{K}$, and the *guessing entropy* of k^* is $\boldsymbol{r}_{N_a}[k^*]$, or r_{k^*}. Because of the repeated attacks, we acquire multiple GE values, which we average to determine the final estimated GE. The $Q_{t_{GE}}$ value is the minimum number of attack traces that are required for the very first guess of the model to be correct, i.e. $\boldsymbol{r}_{Q_{t_{GE}}}[k^*] = 1$, and it can be used to describe the efficiency of the attack model [Rij+21]. In case the available attack traces N_a are not sufficient, this value is not well-defined, but for the sake of being able to perform aggregation in the experiments, we choose to set it to N_a.

2.2 Leakage Models

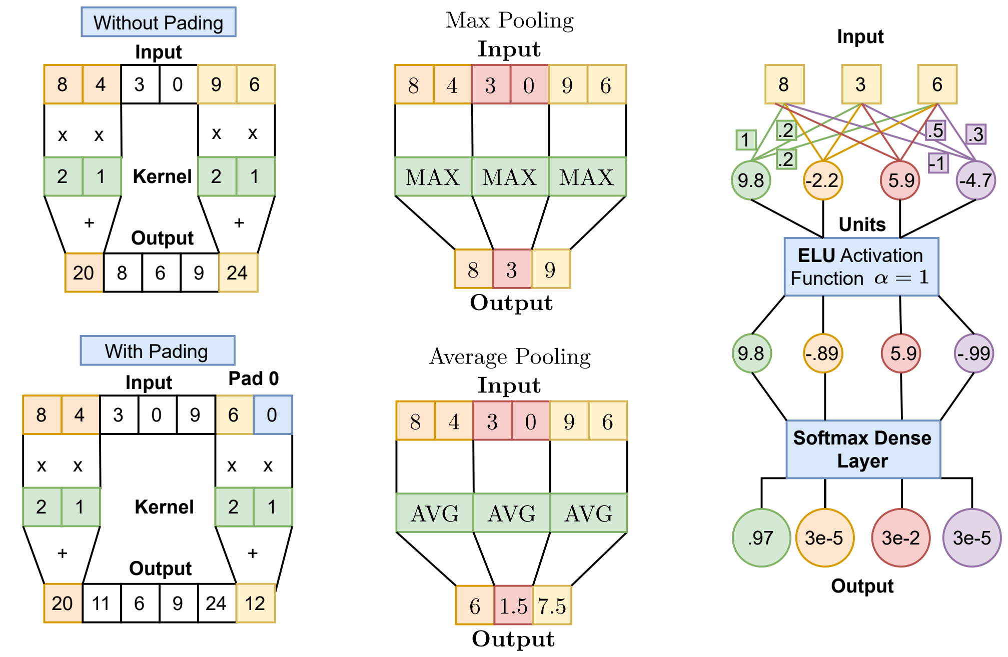
The leakage model defines which information is expected to be leaking from the device in the measurements. Since we focus on AES-128 in our experiments, we assume the output to the S-box function is leaked. Additionally, we only target a single S-box corresponding to a single key byte in the very first execution step of `SubBytes` in the first AES round. We believe our results to apply to other key bytes and later rounds, as determined by [GJS19]. We investigate two types of leakage models for this output byte, the Hamming weight (HW) leakage model and the Identity (ID) leakage model.

Identity Leakage Model In this model, the attacker assumes that the leakage l or power consumption is directly linked to the entire S-box output. For the 8-bit S-box used in AES, this leakage model results in 256 classes representing every possible value of the input byte. The dataset is then labeled with $\phi(p_i, k_i) = sbox(p_i \oplus k_i)$.

Hamming Weight Leakage Model In this model, the attacker assumes that the leakage l or power consumption is directly linked to the number of bits set to 1 in the S-box output, which is equivalent to its hamming weight (HW). For the 8-bit S-box used in AES, this leakage categorizes 256 possible inputs into 9 classes, from 0 bits set to 8 bits set. This is done with the labeling function $\phi(p_i, k_i) = HW\left(sbox\left(p_i \oplus k_i\right)\right)$. This causes several outputs to map to the same class, since e.g. the output values 1 and 4 both belong to HW class 1, and the full output value cannot be recovered. Using the redundant information over several S-boxes and `SubBytes` rounds, as well as the relationship between them, this still allows full key recovery, as for example demonstrated in the CHES 2018 CTF challenge by the AGSJWS team [GJS19]. Choosing this leakage model produces a large class imbalance because while only a single output maps to class 0 and 8 each, 70 outputs map to class 4. This can have a large effect on a machine learning process and may require custom metrics to account for the imbalance [Pic+18].

2.3 Convolutional Neural Networks

A Neural Network consists of a series of interconnected layers containing Neurons that connect an input layer that is activated according to observation with an output layer corresponding to the prediction of the model for this observation. The structure of these interconnections as well as the method of layer operation can vary significantly and defines the overall Neural Architecture. The MLPs is a very simple Neural Network,



(a) Convolutional Operation

(b) Pooling Operations

(c) Dense Layers

Figure 1: Convolutional Operation, Pooling Operation and Dense layers of CNN

only employing fully connected, or "dense", layers. These were shown to perform SCA efficiently in case there are no countermeasures applied by the system, but often fail for more challenging tasks [Ben+20; MPP16].

In recent work, the CNNs have proven to be very effective in learning a multiclass classification model and breaking a system via hardware side-channels, even if such a system implements countermeasures [Zai+19]. CNNs have shown to be very robust towards the most common countermeasures, namely masking [MPP16; MDP19] and desynchronization [CDP17]. A CNN contains convolutional and pooling layers in addition to dense layers as shown in Figure 2. A CNN can be viewed as an MLP where only each neuron of the layer l is connected to a set of neurons of the layer $l-1$, therefore can perform all the operations that can be performed by an MLP [Kle17; Zai+19]. In addition to that, the CNN architecture imposes inductive biases that are useful for many important applications and that the MLP networks would have to learn [Kle17; Zai+19]. A recent study has shown that overall, CNNs are more efficient and better suited to perform SCA on hardware datasets than MLPs [Cha+22], which is why we chose to focus on different CNN architectures.

The convolutional block consists of the convolutional layer and a pooling layer and the dense block consists of the dense (fully-connected layer). The batch normalization operation is typically applied after the convolutional layer and dense layer. Each layer has some trainable parameters $\hat{\boldsymbol{w}}$ which are used to get the final target function f (c.f. Section 2.1) and some hyperparameters. The hyperparameters are configuration variables of the layer external to the learning model (f) and hugely influence finding an optimal target function f. Now, we will briefly describe the operations performed by these layers.

Convolutional Operation This operation basically re-estimates the value of the input value, by taking a weighted average of the neighboring values as shown in Figure 1a. The weights are defined using a kernel of some size w_k (w_k for 1-D data or $w_k \times w_k$ for 2-D data) and these weights are learned using back propagation algorithm [Zai+19]. This kernel is shifted over the input data (1-D vector or 2-D maps) with a stride until the entire data is covered. The convolutional operation is performed for every shift and produces a weighted average value. Typically this operation is applied multiple times using different kernels and this number is called the filter size f_i of the convolutional layer. If this operation is applied without padding, then the dimensionality of output decreases, and this operation is called the valid padding operation. In order to preserve the dimension, the data is padded with 0, and this operation is called same padding [Zai+19].

The number of trainable parameters for convolutional layers are $[in \times f_i \times w_k \times out] + out$ for 1-D data and $[in \times f_i \times w_k \times w_k \times out] + out$ for 2-D data, where *in* denotes the number of inputs and *out* denotes number of outputs [Rij+21]. The two hyperparameters which need to be searched for an optimal CNN architecture are the kernel size and number of filters for each convolutional layer as listed in Table 1.

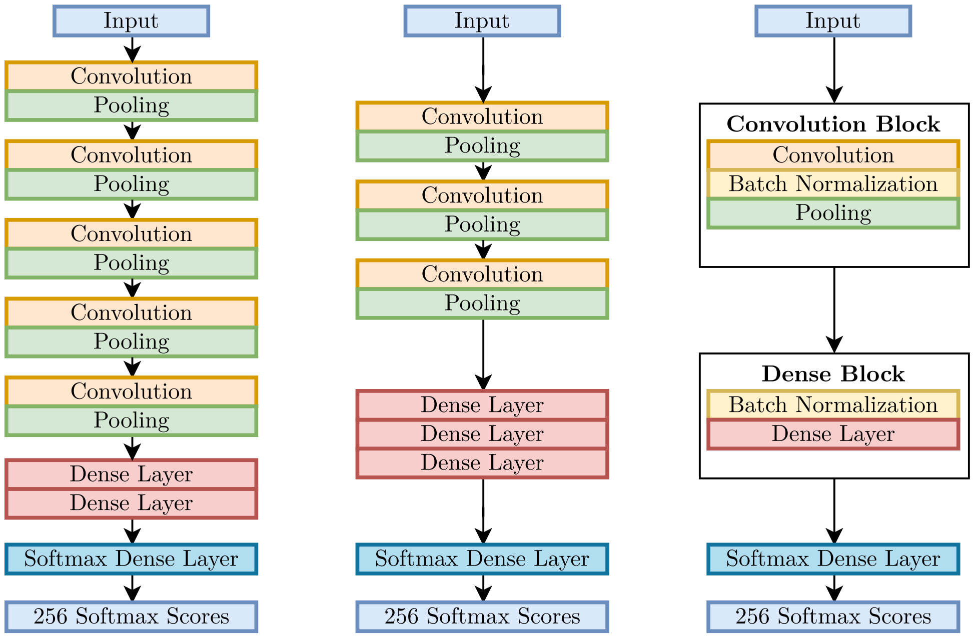
Pooling Operation This operation applies down-sampling on the input acquired from the previous layer and produces a condensed representation. This operation reduces the number of trainable parameters of the CNN and avoids over-fitting [Zai+19]. The pooling operation of some size w_p (w_p for 1-D data or $w_p \times w_p$ for 2-D data) and stride, is shifted across the input and reduces it by applying a max operation or an average operation as shown in Figure 1b. Similar to convolutional operation, pooling operation could also be applied with (preserves dimensionality) or without padding (dimensionality decreases) and the operations are called same or valid padding respectively. This layer does not have any trainable parameters and the hyperparameters which need to be searched for an optimal CNN architecture are the poolsize w_p, number of strides, and pooling operation type as listed in Table 1.

Dense Layers This layer consists of weights $\boldsymbol{W} \in \mathbb{R}^{d \times n_h}$ and biases $\boldsymbol{b} \in \mathbb{R}^{n_h}$, where d is the dimensionality of the input $\boldsymbol{x} \in \mathbb{R}^d$ and n_h is the number of hidden units of the layer [Ben+20]. The output of this layer evaluated using the formulae $\boldsymbol{Wx} + \boldsymbol{b}$ as shown in Figure 1c. Typically, an activation (e.g. ReLU, Elu) is also applied to each element of the output and the weights and biases are learned using the back-propagation algorithm [Ben+20]. The last dense layer is applied using softmax function (c.f. Section 2.1) which converts real-valued scores to *softmax-scores* as shown in Figure 1c. The number of trainable parameters for dense layers is the sum of n_h for each input plus the bias b: $n = (in \times n_h + n_h \times out) + (n_h + out)$ where *in* denotes the number of inputs and *out* denotes number of outputs [Rij+21]. The hyperparameter which needs to be searched is the number of hidden units for each dense layer as listed in Table 1.

Batch-Normalization Layer This layer was introduced to lower internal covariance shift in neural network and thus making the convergence faster [IS15]. It was possible to use larger learning rates for the training process. This layer normalizes every data point x_i in a training batch by estimating the expected mean and the variance of the training batch. The number of trainable parameters for batch-normalization is $4 \times d$, where d is the dimensionality of the input. For NAS, we can choose to either apply it or not in each convolutional block and in each dense block.

2.4 Neural Architecture Search

The first handcrafted CNN architecture is shown in Figure 2a, which was proposed to attack the ASCAD dataset. This architecture was later optimized manually to produce dataset-specific smaller architectures (for example c.f. Figure 2b produced for attacking



(a) ASCAD baseline [Ben+20] (b) Zaid baseline [Zai+19] (c) Our NAS base architecture

Figure 2: Architecture comparison of handcrafted reference CNNs with our NAS structure of generic building blocks

ASCAD_f 50ms and ASCAD_f 100ms). This shows that an optimal CNN architecture is dependent on the dataset and designing it manually requires expert knowledge. This challenge is not unique to side-channel attacks, and consequently, there have been recent developments in automating this process by employing "Neural Architecture Search". NAS treats the task of finding a suitable architecture for a given dataset as a simple optimization problem (using objective) that can be solved automatically. NAS takes a search space $\mathcal{A}$ containing possible architectures and the dataset as input and, using a specified search strategy, automatically searches for the optimal architecture as shown in Figure 3. Typically, NAS uses an evaluation metric, e.g. accuracy or a loss function as its *objective*, which is used as a criterion to evaluate or measure the performance of an architecture. The dataset is split into training data $\mathcal{D}_{train}$, which is is used for training a new architecture $A \in \mathcal{A}$ and validation data $\mathcal{D}_{val}$ which is used to evaluate the performance A. In the end, the NAS produces the best-performing architecture according to the defined *objective* [EMH19]. This motivated the usage of NAS approach, which takes the profiling dataset as input and automatically produces an optimal CNN architecture to perform the SCA for a given dataset [EMH19].

Previous Proposals Recent works propose using NAS for SCA [WPP20; Rij+21]. They first proposed different white-box metrics for defining the objective for performing the NAS using Random and Bayesian search strategy [WPP20]. These white-box metrics determine the cumulative score of the secret key byte on the labeled attack dataset in order to evaluate the performance of an architecture. They extended their work by proposing a novel reinforcement learning based NAS approach which uses the white-box objective as the reward function for learning the Q-function [Rij+21]. The Q-function is used to guide the search and choose the hyperparameters of the next architecture to be evaluated.

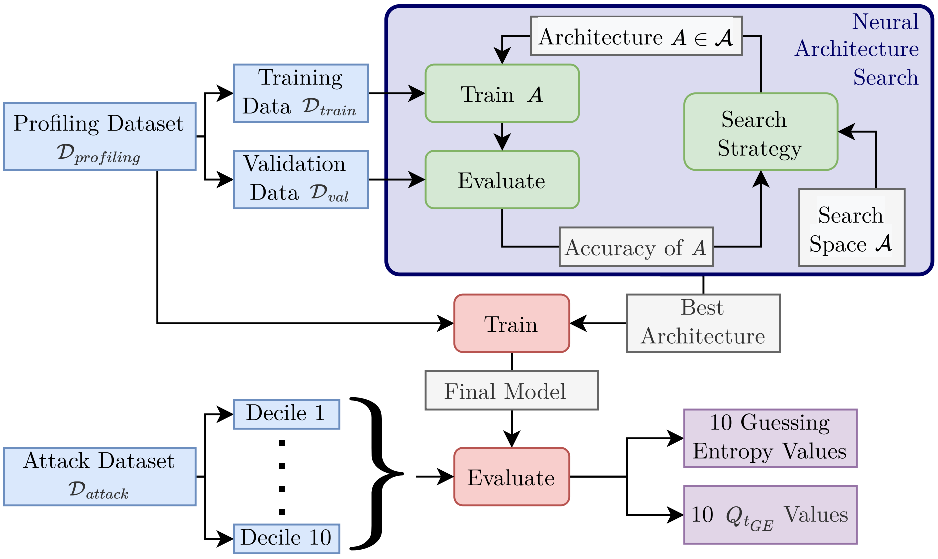
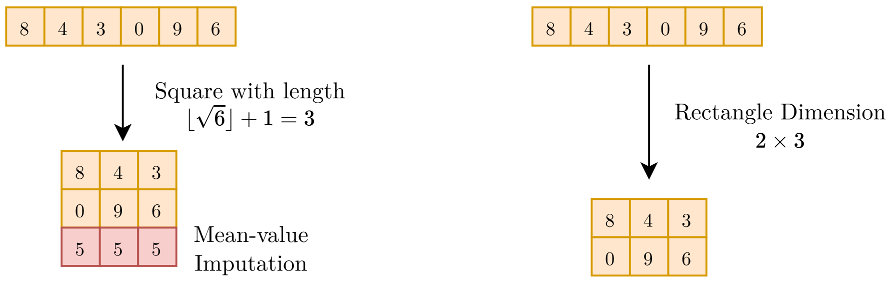

Neural Architecture Search
Architecture $A \in \mathcal{A}$
Profiling Dataset $\mathcal{D}_{profiling}$
Training Data $\mathcal{D}_{train}$
Validation Data $\mathcal{D}_{val}$
Train A
Evaluate
Search Strategy
Search Space $\mathcal{A}$
Accuracy of A
Train
Best Architecture
Final Model
Attack Dataset $\mathcal{D}_{attack}$
Decile 1
Decile 10
Evaluate
10 Guessing Entropy Values
10 $Q_{t_{GE}}$ Values

Figure 3: Schematic of our NAS approach for black-box attacks

This approach uses guessing entropy and $Q_{t_{GE}}$ value of the secret key byte k^* of the system, to evaluate its white-box objective or the reward function. The drawback of these approaches is that it uses the attack dataset to evaluate the objective function to find the architecture, which poses two major issues. First, it no longer allows the detection of overfitting, e.g. where a model is specifically matching only the data it has been exposed to before, performing exceptionally well on training data, but the model does not generalize, meaning that its performance with any other data is poor. Since the attack dataset is also used for hyperparameter-optimization, the architecture is specifically fitted to the attack dataset. To detect overfitting, a hold-out dataset, which is deliberately withheld during the entire model selection, parameter tuning and training process, has to be used, since only the performance on this hold-out can predict the generalization capabilities of the model. If this hold-out dataset is used in any part of the process, even if it was just manually inspected to select which specific classifier to train on it, data-snooping occurs and it is no longer useful for assessing generalization [Jen00]. This is the case for [Rij+21] and [WPP20], and therefore we cannot be sure if overfitting occurred in these experiments. Second, testing a model on the same attack dataset for which its architecture has been optimized will necessarily result in an over-estimation of its real-world attack performance where the architecture cannot be optimized for the unknown key in the attack dataset. This is acceptable in a white-box or gray-box setting where some parts of the attacked device are assumed to be known, but is not compatible with the black-box setting we assume, where the attack dataset would be considered unlabeled for the purposes of training. The performance results reported in [Rij+21] and [WPP20] are thus not necessarily representative of real-world performance on an unlabeled attack dataset and cannot be meaningfully compared to our work.

3 Our Approach

We aim to produce an unbiased, optimal CNN architecture in a black-box setting with the help of NAS. Since NAS was designed primarily to use only the training dataset for finding an optimal architecture, we devised an approach, illustrated in Figure 3, which can



(a) Square Input Conversion

(b) Rectangle Input Conversion

Figure 4: Conversion technique of 1-D input to 2-D Square input and 2-D Rectangle input

satisfy these black-box requirements. This follows the standard training-test-validation split used in many ML methods. For this the profiling dataset is split into validation and training dataset, such that the training dataset $\mathcal{D}_{train}$ is used to train the architecture under consideration and validation data $\mathcal{D}_{val}$ is used to estimate the performance of said architecture. In our approach, the search strategy uses a predefined search space $\mathcal{A}$ (c.f. Table 1) containing CNN architectures (1-D or 2-D depending on the input shape) to perform NAS. Typically, the search strategy initially suggests some architectures $A \in \mathcal{A}$ randomly (exploration) and uses their evaluated accuracy to make future suggestions (exploitation). The training data ($\mathcal{D}_{train}$) is used for training the architectures A and the validation data ($\mathcal{D}_{val}$) is used for evaluating the accuracy of A. Since we are only using the profiling dataset for searching and evaluating the architecture $A \in \mathcal{A}$ under consideration, this makes our NAS optimization a *black-box* approach [Rij+21]. This is in contrast to Wu, Perin, and Picek [WPP20] and Rijsdijk et al. [Rij+21], which use the test dataset instead of a validation dataset for guiding the search, making their approach *white-box* since the test dataset can then not be used to give an unbiased performance evaluation. In the end, NAS suggests the architecture which has the highest accuracy. The best-performing architecture is then trained on the complete profiling dataset ($\mathcal{D}_{\text{profiling}}$), which improves the performance while only incurring a marginal computational overhead compared to the actual search. Similar to the folds used in cross-validation, the attack dataset ($\mathcal{D}_{attack}$) is split into 10 equal parts (deciles) and each part is used to evaluate the attack efficiency of this model using the *guessing entropy* and $Q_{t_{GE}}$ measures defined in Section 2.1.

3.1 Two-Dimensional Input Reshaping

The measurement traces of the datasets have to be transformed into the proper shape for the neural network to process. The most straightforward transformation is to produce a One-Dimensional input from the time series input and define a search space containing 1-D CNN architectures (c.f. Table 1) and to apply NAS to find an optimal architecture. While most of the 1-D CNN architectures that are explored for performing SCA to break AES-128 encryption were originally inspired by popular 2-D CNN architectures proposed for image classification, like VGGNet, Inceptionv3 [Ben+20], they are only applied on one-dimensional inputs. Recent work has shown that using 2-D CNNs could increase the accuracy and efficiency for breaking post-quantum key-exchange (PQKE) protocols [Kas+21; Het+20]. This motivates us to also explore using two-dimensional inputs for AES-128 attacks, applying NAS on a search space containing 2-D CNN architectures (c.f. Table 1). In order to convert the one-dimensional input to two-dimensional input, we propose to use two techniques Square and Rectangle. In the Square approach, we create a square of dimension $\lfloor\sqrt{d}\rfloor + 1$ and fill the remaining places with the mean value of the instance as shown in Figure 4a. For forming a Rectangle input, we try to create the rectangle that is closest to a square but does not require filling with imputed values. As shown in Figure 4b, these dimensions can be determined using the two factors of d which are closest to $\sqrt{d}$.

3.2 Search Strategies

The previous work on applying NAS for performing SCA only considered Bayesian and Random search strategies [WPP20], which are not necessarily time-efficient [EMH19; Li+17]. We explore the four search strategies Random, Greedy, Hyperband, and Bayesian with 1000 fixed trials and use their respective implementations provided by AutoKeras [JSH19].

Typically, search strategies first explore the search stage by trying many substantially different architectures. This is followed by exploitation, in which well-performing architectures are further improved via small changes. Because each search algorithm is only allowed to consider a limited number of architectures (in our case 1000), it should have a balanced trade-off between exploration and exploitation in order to perform efficiently and accurately.

Random The Random search technique chooses a unique architecture configuration uniformly at random (with replacement) from the complete search space and evaluates its performance. This process is repeated for a specified number of trials but is preempted if it has exhausted the search space or the same configuration is chosen multiple times. This means it focuses only on exploration without any exploitation.

Greedy The Greedy search algorithm proposed by Jin [Jin21] works in two distinct stages, exploration and exploitation. In the exploration stage, it evaluates uniformly randomly chosen models for a limited number of trials [JSH19]. In the exploitation stage, it generates models which are neighboring the best-performing model from the first stage and exclusively try to improve this model. This is done by traversing a hierarchical hyperparameter tree representing the hyperparameters of the best-performing model, as well as possible changes such that the new hyperparameter values remain close to that of the best-performing model with high probability [Jin21]. This tree is rebuilt if a better architecture is found, at which point this architecture becomes the starting point for subsequent exploitation. The algorithm continues the exploitation process until either the trial limit is reached or until it has exhausted the entire search space. The Greedy search algorithm is time-efficient, but it might get stuck in local optima since it performs limited exploration, e.g. for only 1 % of the maximum number of trials in `AutoKeras` [Jin21; JSH19].

Hyperband The Hyperband search algorithm is based on the successive halving algorithm [Li+17]. The Successive Halving algorithm divides the resources (time, epochs) equally into a specific number of hyperparameter configurations. In each time step (2-3 epochs) it checks their performance and at the end keeps the top-half best-performing configurations. This process is repeated until only the best-performing configurations are left. The Hyperband algorithm is time-efficient and balances the trade-off between exploration (check many models with a low budget) and exploitation (provide a high budget to the best-performing architectures) very well [Li+17].

Bayesian The Bayesian search algorithm is based on Bayesian optimization, which assumes a (black-box) function $f' : \boldsymbol{\Theta} \to \mathbb{R}$ over the hyperparameter space $\boldsymbol{\theta} \in \boldsymbol{\Theta} = (\Theta_1, \ldots, \Theta_n)$, such that $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$ represents one hyperparameter configuration [FSH15].

Each hyperparameter space can be an integer, real or categorical, i.e., $\Theta_i \in \mathbb{R}$ or $\Theta_i \in \mathbb{Z}$ or $\Theta_i \in \{0, \ldots, c\}$, for some $c \in \mathbb{N}$, where c is the number of categories. For finding the best configuration function f', which maps a configuration $\boldsymbol{\theta}$ to the estimated real-valued validation accuracy, a probabilistic model using Gaussian Process, is used. This probabilistic model provides the properly balanced trade-off between exploitation and exploration of the search space [Li+17]. At the end of the 1000 fixed trials the best configuration is obtained as $\theta^* = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} f'(\theta)$.

4 Setup of Our Parameter Study

The performance of the final model returned by performing NAS as described in Section 3 heavily depends on two factors or parameters: First, the search strategy employed for NAS, which has a huge influence both on search performance and runtime, and second, the shape of the input features. In order to determine the best-performing options for both of these factors, we need to perform a parameter study. The methodology for this study will be outlined in Section 4.1, while Section 4.2 presents the datasets we used for the experiments and Section 4.4 covers the hardware necessary for it. Because we investigate how our automated NAS setup compares to manually crafted baseline architectures, we additionally trained the fixed CNN architectures we describe in Section 4.3.

4.1 Methodology

We proposed a black-box NAS approach to automatically perform a SCA as described in Section 3. The two main phases of our empirical study are first analyzing the expected success rate of our approach for a given parameter combination of the search strategy and input shape. Using this, we will achieve the best parameter combination of the search strategy and input shape, which should be used by NAS to perform SCA. In addition, we would also like to compare the attack performance of the CNN architectures produced by the NAS configured with the best parameter combination of the search strategy and input shape with the state-of-the-art handcrafted CNN architectures.

Parameter Study The attack performance of the final architecture produced by the NAS is heavily dependent on the search strategy used and the input shape of the data. For the parameter study, we consider 4 search strategies, namely RANDOM, GREEDY, HYPERBAND and BAYESIAN (c.f. Section 3.2). Our input data could be ONE-DIMENSIONAL (1-D CNN), rectangular (2-D CNN) or square (2-D CNN) shaped (c.f. Section 3.1). This gives us a total of 12 parameter settings for using the NAS to perform SCA, e.g. using HYPERBAND search strategy on rectangular input data. In order to perform a study, for each parameter setting we apply NAS on 10 dataset (c.f. Section 4.2) on 2 leakage models (HW and ID, c.f. Section 2.2). For each parameter combination (input shape and search strategy), dataset, and leakage model, we apply the NAS to acquire the best-performing model. This model is then trained on a complete profiling dataset $\mathcal{D}_{\text{profiling}}$ and evaluated on 10 equal parts (deciles) of the attack dataset $\mathcal{D}_{attack}$. In the end, we acquire 10 guessing entropy values and 10 $Q_{t_{GE}}$ values, which are aggregated to achieve the final performance of the parameter combination on the dataset.

Loss Function Investigation We initially intended to also investigate the influence of the loss function in our parameter study, considering the most commonly used loss function CCE and comparing it to 6 other specialized loss functions, among them Ranking Loss (RKL) and Focal-Loss Categorical Cross-Entropy (FLCCE) [Ker+22]. The parameter study was set up to combine all possible combinations of 3 input shapes, 4 search strategies, and 7 loss functions, and then run on the supercomputer (c.f. Section 4.4). When analyzing the results, we discovered that a small bug in the ML library we used, `AutoKeras`, prevented the loss function parameter to be passed onto the actual CNN training module, forcing all of the experiments to use the default value CCE instead. We therefore effectively evaluated our NAS approach for the same choice of input shape and search strategy 7 times, using CCE as the loss function each time. Each of the 7 runs produced a different split of profiling dataset $\mathcal{D}_{\text{profiling}}$ into training $\mathcal{D}_{train}$ and validation $\mathcal{D}_{val}$ dataset, resulting in a different best architecture returned by NAS and a different final model. These experiments had already been executed and consumed significant resources, so we had to consider if the results should be outright discarded or not. Because of the randomization involved, it is necessary to run such an experiment several times and average the results to get a statistically significant and "realistic" estimation of performance[1] [LT20]. Consequently, we decided to keep all of the original experiments and aggregate the results of the 7 models as independent repetitions of the same NAS experiment. Unfortunately, we will therefore not be able to determine the impact of the loss function.

Evaluation of Single Experiment Run A single experiment run for our study consists of applying NAS configured with a unique combination of input shape, search strategy, dataset, and leakage model. For each experiment, there are a total of 70 attacks being executed: 7 different models are trained based on unique random seeds, then the attack is executed for each of the 10 attack deciles for each model. The results of the 70 attacks are then aggregated to determine the performance with the following metrics.

Success Rate A single attack is considered to be successful if the final guessing entropy is 1, e.g. the correct key is at the top of the predictions with $r_{k^*} = 1$ [TPR13]. The success rate is defined as the empirically determined probability of an attack being successful. For our 70 attacks, we can therefore count the number of successful attacks and divide by the total number of attacks [TPR13], returning a percentage value.

Attack Efficiency $Q_{t_{GE}}$ Even though several approaches might achieve a high success rate, it is still preferable for them to use fewer attack traces, making them more efficient. This can be measured by the $Q_{t_{GE}}$ value, which counts the number of attack traces that are required to achieve a successful attack. To calculate the $Q_{t_{GE}}$ for an experiment run, we can average the $Q_{t_{GE}}$ values of the 70 individual attacks. This value can be sensitive to outliers, e.g. unsuccessful attacks where $Q_{t_{GE}}$ is set to N_a.

Evolution of Guessing Entropy Another analysis method of the model performance involves plotting the development of the GE value "over time", after observing a certain number of attack traces. As detailed in Section 2.1, this GE is already the result of running the attack 100 times on shuffled attack traces. For our analysis, we aggregate the 10 attack runs by averaging and reporting the 7 NAS models individually.

Implementation Details To implement NAS for different search strategies and search spaces as well as input shapes, we extend the `AutoModel`, `DenseBlock`, `ConvBlock` and `ClassificationHead` class from the `AutoKeras` python library [JSH19], a popular tool for NAS. The code for the experiments and the generation of plots with detailed documentation is publicly available on GitHub[2].

Architecture Search Space To perform NAS, we need to define the search space depending on the input shape of NAS. In Table 1, we define a search space containing

[1] As recommended in https://github.com/keras-team/autokeras/issues/359

[2] https://github.com/prithagupta/deep-learning-sca

Table 1: Overview of the Search Space for our NAS approach

Hyperparameter Type	Hyperparameter	Possible Options
Whole Network	Optimizer	{'adam' , 'adam_with_weight_decay'}
	Learning rate	{1e-1, 5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5}
Every Layer	Dropout	{0.0, 0.1, 0.2, 0.3, 0.4, 0.5}
	Use Batch Normalization	{True, False}
	Activation Function	{'relu', 'selu', 'elu', 'tanh'}
Convolutional Block	# Blocks	{1, 2, 3, 4, 5}
	Convolutional Kernel Size	{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
	Convolutional Filters	{2, 8, 16, 32, 64, 128, 256}
	Pooling Type	{'max' , 'average'}
	Pooling Strides 1-D CNN	{2, 3, 4, 5, 6, 7, 8, 9, 10}
	Pooling Poolsize 1-D CNN	{2, 3, 4, 5}
	Pooling Strides 2-D CNN	{2, 4}
	Pooling Poolsize 2-D CNN	Convolutional Kernel Size-1
Dense Block	# Blocks	{1, 2, 3}
	Hidden Units	{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}

different configurations for 2-D CNN architectures for Rectangle and Square inputs and a search space containing different configurations for 1-D CNN architectures for One-Dimensional input. As described in Section 2.3, the hyperparameters for the convolutional layer are the kernel size and the number of filters, the pooling layer is the poolsize w_p, the number of strides and pooling operation type and for a dense layer is the number of hidden units. We selected the range for each hyperparameter by analyzing the related work [Rij+21; WPP20]. The search space includes all possible 1-D baseline CNN architectures [Zai+19; Ben+20], and as such the search strategies could in theory find these fixed architectures and match their performance. In particular, Random is certain to find these architectures eventually. We also included the range of each hyperparameter from the search space designed for 1-D CNN architectures proposed by the current work done on NAS for performing SCA [Rij+21; WPP20]. If the characteristics of the dataset are known in advance, it is possible to reduce this search space to make the search more efficient. We chose not to tailor the search space to the dataset like this, as this way the architecture design remains fully automated, requiring no manual analysis of the dataset.

The network, layer, and dense block hyperparameters are the same for both 1-D CNN search space and 2-D CNN search space. Additionally, the range of convolutional kernel size, convolutional filters, and pooling types for each convolutional block is also the same for both search spaces. To avoid the formation of invalid 2-D CNN architectures, the range of pooling strides is smaller and the poolsize value is set using the kernel size, which is the suggested default of `AutoKeras` [JSH19]. This makes the search space relatively smaller for 2-D CNNs. The padding type is set to the "same" padding type for convolutional layer and "valid" padding type for pooling layer for 1-D CNNs search space. To avoid the formation of invalid 2-D CNN, the padding type for the convolutional and pooling layer is set using the kernel size of the convolutional layer, using the formulae proposed by `AutoKeras` [JSH19]. The total number of possible hyperparameter configurations for 1-D CNNs is 40 758 681 600 and for 2-D CNNs is 2 264 371 200. For our experiments, we set the maximum number of trials to 1000 , which implies that only 1000 possible architectures are explored by NAS out of such a large search space. Each search strategy is provided with the same budget limit, which means that only around 5×10^{-5} % of 2-D search space and 2.5×10^{-6} % of 1-D search space is explored in order to produce an optimal CNN architecture.

4.2 Datasets

We want to investigate the behavior of NAS in a wide range of settings using well-known datasets to enable direct comparisons to other works in hardware side-channel attacks. We, therefore, chose to focus on five datasets that have already been investigated before: ASCAD v1, DPA contest v4.1, AES_RD, AES_HD, and CHES CTF 2018 AES-128. All of these datasets record implementations of AES-128, which means the same overall approach should work automatically for each of them, without any need for dataset-specific tuning. However, the systems incorporate different types of leakage countermeasures, e.g. masking or random delays. The configurations for all these datasets are listed in Table 2.

ASCAD The ASCAD dataset was proposed as a benchmark dataset, containing EM radiation measurements obtained from an ATMega8515 (8-bit microcontroller with AVR architecture) device running a masked implementation of AES-128 [Ben+20]. This dataset is available in different versions and we utilize the original v1 dataset in both fixed key (ASCAD_f[3]) and variable key (ASCAD_r[4]) variants. The traces in these datasets have been aligned such that the AES computation always starts at the same sample within each trace. The points of interest in the raw data have been analyzed using the signal-to-noise ratio and only a small subset of samples from the full traces is used [Ben+20]. Additionally, we consider versions that add random amounts of desynchronization to each trace. These versions were created by Benadjila et al. to simulate imprecise temporal alignment of the traces by adding artificial jitter to each trace. The traces have been desynchronized by a maximum of 50 as well as 100 samples, resulting in the 6 different varieties of the ASCAD dataset listed in Table 2.

CHES CTF The CHES CTF dataset consists of traces originally produced for the CHES 2018 AES-128 CTF challenge. Note that the dataset we consider[5] is a single reduced version (45 000 traces) derived from the original measurements (500 000 traces) and not identical to the datasets published as part of the actual contest[6], which consists of 6 different sets (42 000 traces). This reduced version is used in the AISY framework and has already been pre-processed [PWP21].

AES_RD AES_RD[7] was originally used to investigate random delay countermeasures [CK09]. The traces for this dataset are collected from an 8-bit ATMEL AVR

[3] https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1/ATM_AES_v1_fixed_key/

[4] https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1/ATM_AES_v1_variable_key/

[5] http://aisylabdatasets.ewi.tudelft.nl/ches_ctf.h5

[6] https://chesctf.riscure.com/2018/content?show=training

[7] https://github.com/ikizhvatov/randomdelays-traces/

Table 2: Details of the datasets considered

Dataset name	# Features	# Profiling traces	# Attack traces	Attack byte
ASCAD_f	700	50000	10000	2
ASCAD_f desync50	700	50000	10000	2
ASCAD_f desync100	700	50000	10000	2
ASCAD_r	1400	50000	100000	2
ASCAD_r desync50	1400	50000	100000	2
ASCAD_r desync100	1400	50000	100000	2
CHES CTF	2200	45000	5000	2
AES_HD	1250	50000	25000	0
AES_RD	3500	25000	25000	0
DP4CONTEST	4000	4500	5000	0

microcontroller running an AES-128 implementation incorporating random delays. We use the converted dataset[8] as analyzed in [Zai+19].

AES_HD AES_HD[9] dataset contains EM measurements obtained from Xilinx Virtex-5 FPGA (coded in VHDL) implementing an unprotected AES-128 implementation [Pic+18]. A big difference in this dataset is the fact that it records the AES decryption operation instead of the encryption operation. The labels are generated for a difference leakage model based on the ciphertext bytes c_i^j used in the decryption, specifically the 12th (c_i^{11}) and 8th (c_i^7) ciphertext bytes. The resulting label is then calculated with $\phi(c_i, k_i) = sbox^{-1}(c_i^{11} \oplus k_i) \oplus c_i^7$ (c.f. section 2.1). In our Hamming weight experiments, taking the Hamming weight of this label results in the Hamming Distance (HD) leakage model used in [Pic+18]. We again use the converted dataset[10] as analyzed in [Zai+19].

DPAv4 The DPAv4 dataset[11] contains traces obtained from ATMEL AVR-163 microcontroller running an AES-128 implementation protected with Rotating Sbox Masking (RSM) [Bha+14]. It was used in the fourth edition of the DPA contest, from which we only consider the “improved” masked AES-128 target contained in dataset version 4.2. We again used the extracted dataset[12] from [Zai+19]. In order to be consistent while comparing our approach with the performance of the baselines proposed by [Zai+19], we assume that the mask value is known, essentially nullifying the masking.

4.3 Baseline Architectures

We also need to train fixed baseline architectures for comparison with our NAS approach. We chose to use the ASCAD architecture as proposed in [Ben+20] and ZAID architectures as proposed in [Zai+19]. The ASCAD baseline is the CNN model inspired from the image recognition baseline VGG-16 CNN model [SZ15] and shown in Figure 2a. Zaid et al. proposed several specific architectures for ASCAD_f, ASCAD_f 50ms, ASCAD_f 100ms, AES_HD, AES_RD and DPAv4 datasets [Zai+19]. For ASCAD_r, ASCAD_r 50ms, ASCAD_r 100ms there is no specific proposal, so we use the corresponding baselines proposed for the fixed-key versions (ASCAD_f, ASCAD_f 50ms, ASCAD_f 100ms). Since CHES CTF is known to be a very hard dataset, we use the deepest CNN proposed by ZAID [Zai+19], which is the architecture for the ASCAD_f 100ms dataset shown in Figure 2b.

4.4 Computing Hardware

Our experiments necessitate training millions of CNN models in total, which requires thousands of hours of GPU time. We consequently ran them in parallel on a supercomputer equipped with GPU nodes, which allowed us to finish the entire parameter study in a few weeks. These GPU nodes consist of two AMD Milan 7763 CPUs running at 2.45 GHz, 512 GB of main memory, and four Nvidia A100 GPUs equipped with NVLink and 40 GB HBM2 GPU memory. A single NAS experiment, which consists of determining the best architecture through repeated intermediate model training and subsequent final model training, takes less than 2 days on these shared GPU nodes. The training times are mostly

[8] https://github.com/gabzai/Methodology-for-efficient-CNN-architectures-in-SCA/tree/master/AES_RD/AES_RD_dataset

[9] https://github.com/AESHD/AES_HD_Dataset/

[10] https://github.com/gabzai/Methodology-for-efficient-CNN-architectures-in-SCA/blob/master/AES_HD/AES_HD_dataset.zip

[11] http://www.dpacontest.org/v4/42_traces.php/

[12] https://github.com/gabzai/Methodology-for-efficient-CNN-architectures-in-SCA/blob/master/DPA-contest%20v4/DPAv4_dataset.zip

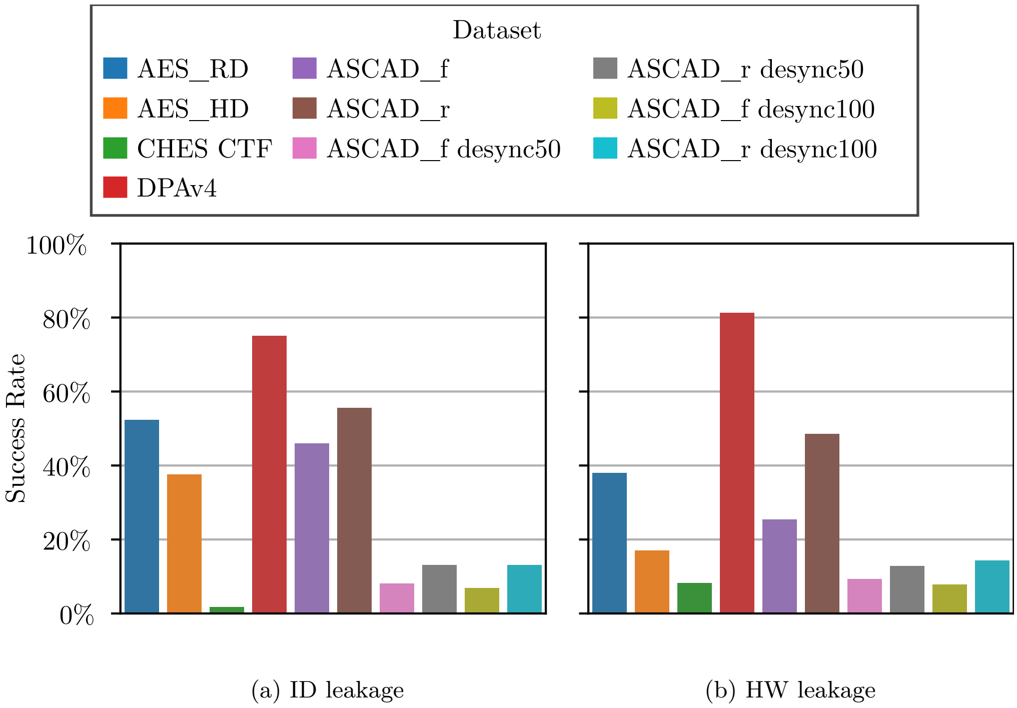
dependent on the search strategy, and to a lesser degree on the dataset. Even though all search strategies were provided with a budget for 1000 trials, only Random and Bayesian search strategies used up the entire budget every time, with one experiment running for at most 2 days. The Greedy strategy used only a portion of the provided budget, which resulted in a maximum running times of 5 hours. In the case of Hyperband strategy, even though it trains 1000 models, it preempts and rejects some of them early due to the successive halving technique, resulting in a reduced maximum running time of 12 hours. However, these experiments were run in a shared environment concurrently with other computations. To determine how long one would have to run a NAS experiment on consumer hardware, we also re-ran an example experiment for the AES_HD dataset on an otherwise idle gaming PC. This $2500 gaming PC is equipped with an AMD Ryzen 9 3950X CPU running at 3.5 GHz, 64 GB of main memory and a single GeForce RTX 2080 Super GPU with 8 GB GDDR6 GPU memory. Our example experiment took 22 hours for Random, 3 hours and 45 minutes for Hyperband, 2 hours and 55 minutes for Greedy, and 10 hours and 12 minutes for Bayesian. We consider 22 hours of runtime on consumer hardware to be a reasonable assumption for the lower limit of computational resources that an attacker might be willing to commit to a single attack, with well-equipped attackers likely far exceeding these constraints.

5 Parameter Study Results

We ran the full parameter study outlined in Section 4, combining the possible options for search strategy and input shape. These were applied to the 10 datasets detailed in Section 4.2, both for an ID leakage model as well as an HW leakage model. Additionally, we trained the baseline architectures described in Section 4.3 for each of the datasets.

5.1 Overall Reliability

First, we want to determine the overall reliability of our NAS approach, so we determined the success rate in all the experiments executed for each dataset. In our context, an attack is considered successful if the final guessing entropy is 1 after processing all the traces in the respective attack dataset decile. We show this per-dataset attack success rate separately for ID (Figure 5a) and HW (Figure 5b) leakage, giving a rough indication of how difficult attacking each dataset is. The first observation is the relative ease with which the DPAv4 dataset can be attacked: Even when taking all the suboptimal combinations of the search strategy and input shape into account, over 75 % of the attacks on it are successful, regardless of leakage model. This is hardly surprising, as the dataset variant we consider effectively contains no countermeasures (see Section 4.2 for details). Another unsurprising result is the increased difficulty incurred by desynchronization: When comparing the non-desynchronized version of ASCAD_f and ASCAD_r to their desynchronized counterparts, the degradation in reliability is clear, although some of the attacks are still able to succeed. We also need to point out the disastrous performance of NAS when applied to the CHES CTF dataset in the ID leakage model, where only a handful of attacks were able to recover the full identity value. When comparing ID and HW leakage, we would expect HW leakage to be more successful, as the models do not need to recover the full 256 identities but only the 9 different Hamming weight classes. This is not the case, evidently since the NAS performance noticeably degrades for datasets like AES_HD and ASCAD_f. This points to a possible issue with using accuracy as the optimizing goal for NAS: Upon manual investigation of the models produced by NAS, a large portion of them would simply predict a HW of 4 regardless of input. Because of the imbalance explained in Section 2.2, this "blind" strategy produces a decent accuracy of over 27 % on random inputs. A NAS search algorithm can get stuck on this local optimum, degrading the overall success rate of an



(a) ID leakage

(b) HW leakage

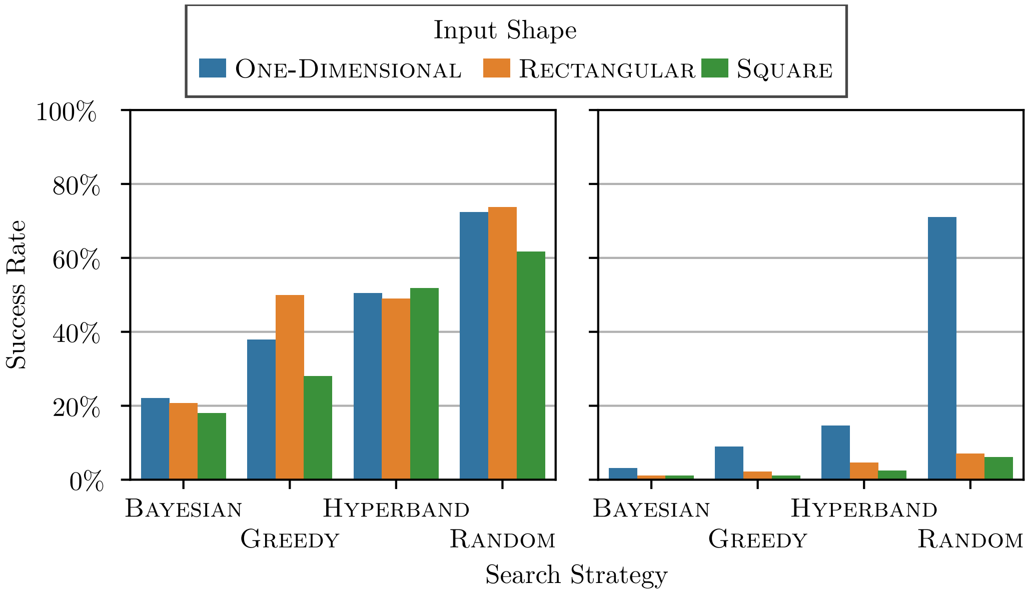
Figure 5: Success rate of overall experiment runs for each dataset in the ID and HW leakage models.

attack.

5.2 Optimal NAS Parameters

In order to answer the second question, we plotted the success rate for every possible NAS parameter combination of input shape and search strategy. We chose to plot them separately for ID and HW leakage models, as well as synchronized and desynchronized datasets because of the large difference in performance.

Identity Leakage Figure 6 shows the influence of the choice of each possible NAS parameter combination (input shape and the search strategy) on the overall performance of synchronized and desynchronized datasets for ID leakage. This clearly shows that the choice of search strategy has a significant impact on the success rate, which rises from around 20 % to around 70 % when going from Bayesian search via Greedy and Hyperband to Random search. Random search strategy clearly outperforms the other strategies, but it comes at the price of being slower, taking about 5 times longer than Greedy and Hyperband (c.f. Section 4.4), mostly because it keeps exploring the search space until the limit of 1000 trials is reached. For synchronized datasets, using the Hyperband strategy might be a viable alternative, as it still produces a CNN with a success rate of around 50 % as shown in Figure 6a, while being substantially faster than Random search. For desynchronized datasets, the difference between Random and its competitors grows even larger, to the point that choosing Hyperband for its speed is no longer a viable option. When comparing input shapes, there does not appear to be a consistent difference between 1-D and 2-D for synchronized datasets, demonstrating the ability of NAS to adjust to vastly different situations. This also shows that while using 2-D CNNs can be a viable option, they fail to provide any improvements compared to the 1-D input shape. This changes



(a) Synchronized datasets

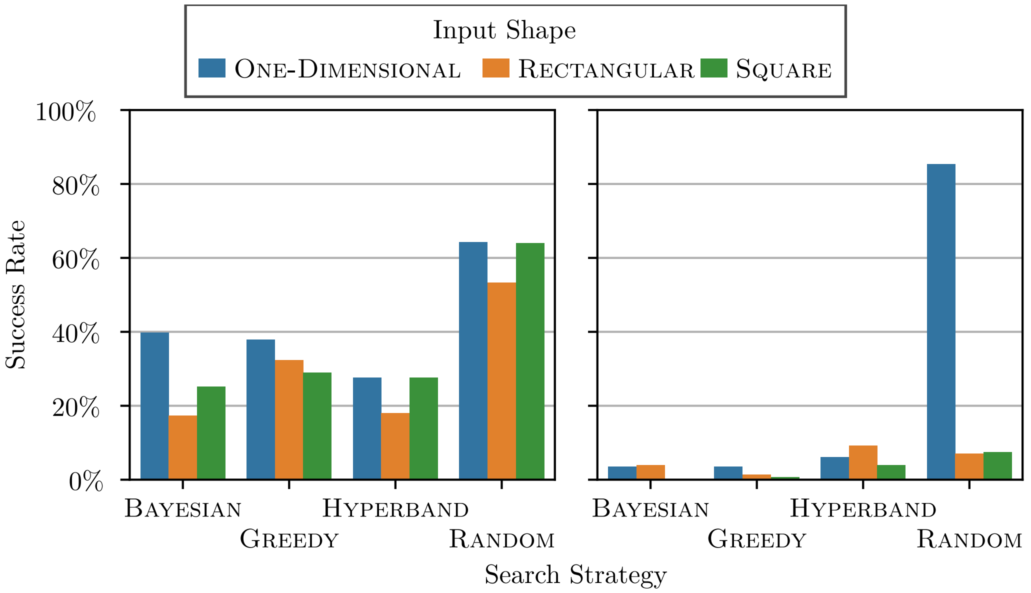
(b) Desynchronized datasets

Figure 6: Influence of NAS parameters on the success rate in the ID leakage model.

drastically when desynchronization gets introduced, which the 2-D CNN architectures are clearly not able to compensate for. Wouters et al. [Wou+20] observed that the first convolutional block removes the desynchronization in the dataset by using the convolutional and pooling operation on neighboring values in the trace. Converting one-dimensional to two-dimensional inputs changes this local relationship between neighboring values, which makes re-synchronizing the traces more difficult.

Hamming Weight Leakage Figure 7 shows the impact of each possible NAS parameter combination on the success rate of the SCA on synchronized and desynchronized datasets for HW leakage. As discussed in Section 5.1, the overall performance of our NAS models is slightly lower compared to ID leakage. When attacking synchronized datasets (as shown in Figure 7a), the trend of increasing success rate from Bayesian via Greedy and Hyperband to Random search can no longer be observed. Only Random performs significantly better than the alternative search strategies, and Hyperband is no longer a viable alternative. Again, the input shape does not appear to play a major role in the success rate on synchronized datasets. When considering only desynchronized datasets, the difference becomes even more pronounced, where only the combination of One-Dimensional inputs with Random search strategy is able to reach any level of success, greatly outperforming all alternative combinations.

Overall Reliability of Optimal NAS Parameters As discussed above, it is clear that using Random search strategy with One-Dimensional inputs gives the highest chances of producing a CNN model which can break the system successfully, especially when desynchronization is involved. We wanted to determine what success rate can be achieved for this specific combination, plotting it per-dataset separately for ID (Figure 8a) and HW (Figure 8b) leakage. The results show that for all other datasets apart from AES_HD and CHES CTF, over 60 % of the attacks are successful, sometimes even with a phenomenal success rate of 100 %. The anomaly of AES_HD and CHES CTF that becomes apparent in this plot will be investigated in detail in Section 5.3. We can conclude that the combination



(a) Synchronized datasets (b) Desynchronized datasets

Figure 7: Influence of NAS parameters on the success rate in the HW leakage model.

of One-Dimensional input shape and Random search strategy appears to be by far the best choice when it comes to reliably creating successful attack models.

5.3 Comparison with Fixed Architectures

As determined in Section 5.2, using the Random search strategy and One-Dimensional input shape yields CNN architectures which can perform SCA with a high success rate. We are also interested in the efficiency of the architectures produced by One-Dimensional and Random, as well as how they compare to traditional fixed architectures.

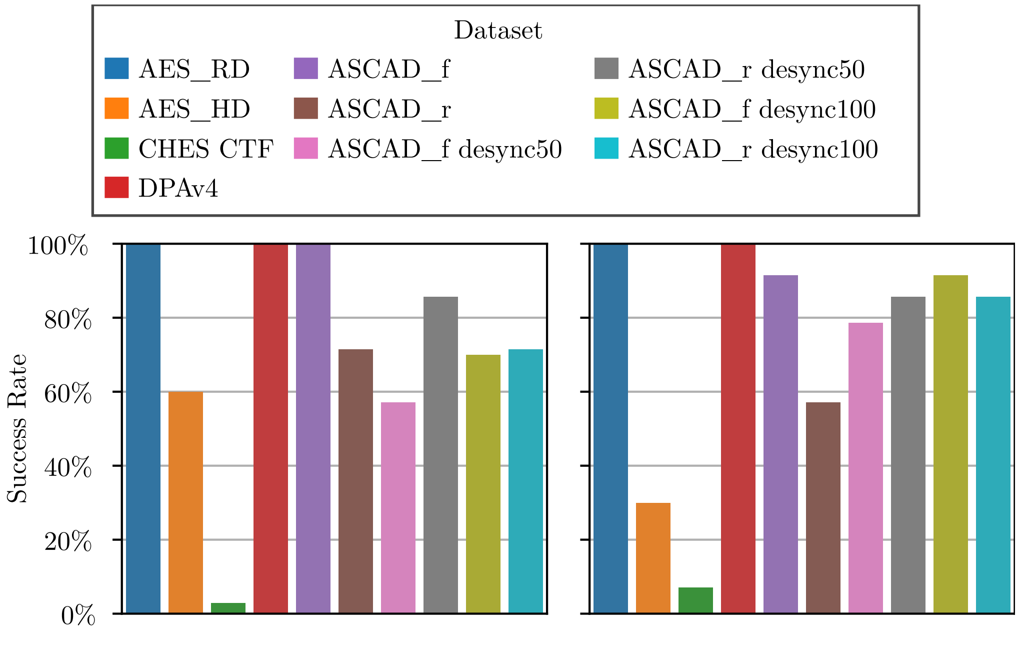
Comparison of $Q_{t_{GE}}$ For the purpose of fairly comparing the efficiency of our approach with the baseline architectures presented in Section 4.3, we are restricting our investigation to ID leakage, since these baselines were only proposed for the ID leakage model. We

Table 3: Comparison of the $Q_{t_{GE}}$ metric for the different datasets on ID leakage

Dataset	ASCAD Baseline	ZAID Baseline	NAS Model
AES_RD	270.3	3.0	**1.39**
AES_HD	2500.0	**443.5**	1463.74
CHES CTF	**498.5**	500.0	499.07
DPAv4	49.3	5.2	**1.11**
ASCAD_f	703.5	1000.0	**127.53**
ASCAD_r	**322.0**	10000.0	2929.77
ASCAD_f desync50	920.9	1000.0	**509.56**
ASCAD_r desync50	4449.0	**40.6**	1553.64
ASCAD_f desync100	974.6	**81.8**	482.57
ASCAD_r desync100	6664.2	**28.3**	2914.01

determined the efficiency of the models with their $Q_{t_{GE}}$ value, which counts the number of attack traces necessary for the model prediction to reach a GE of 1, with lower $Q_{t_{GE}}$ indicating a more efficient attack. We report the average $Q_{t_{GE}}$ for the 7 models NAS produced with Random search strategy on One-Dimensional inputs as well as the model produced ASCAD and ZAID baseline architectures in Table 3. On the “easy” datasets AES_RD and DPAv4 our NAS is able to perform instantaneous attacks, requiring less than 2 attack traces on average. We observe that our NAS models outperform the baselines for 4 out of 10 datasets, but especially the ZAID baselines specializing for desynchronization are much better suited for datasets with desynchronization, where NAS is only able to achieve better efficiency than the ASCAD baseline. However, these specialized models were not able to achieve successful attacks at all for the CHES CTF, synchronized ASCAD, and ASCAD_f desync50 datasets, indicated by the $Q_{t_{GE}}$ matching the total number of attack traces. This indicates some issues when relying solely on the $Q_{t_{GE}}$ for efficiency: The averaged $Q_{t_{GE}}$ is sensitive to outliers and also influenced by the success rate, as unsuccessful attacks will be considered with the total number of attack traces.

Comparison of GE Convergence For the datasets with ID leakage, we therefore also plotted the GE to get a more detailed comparison. Again we compare the evolution of the models produced by NAS using Random search on One-Dimensional inputs against the two baselines and plot their GE convergence in Figure 9. Instead of averaging the 7 NAS models, we plot each model individually and only take the average GE over the 10 attack deciles of each model. This plot reveals that for most of the datasets, the NAS models match the baseline architectures, with the very best NAS model outperforming the baseline in the majority of the datasets (6 of 10). But just considering the best model is not representative of the general performance: On AES_HD, for example, it becomes apparent that while most NAS architectures perform similarly to the ZAID baseline, 3 out



(a) ID leakage

(b) HW leakage

Figure 8: Success rate when selecting One-Dimensional input shape and Random search strategy. Performance for each dataset in the ID and HW leakage models shown.

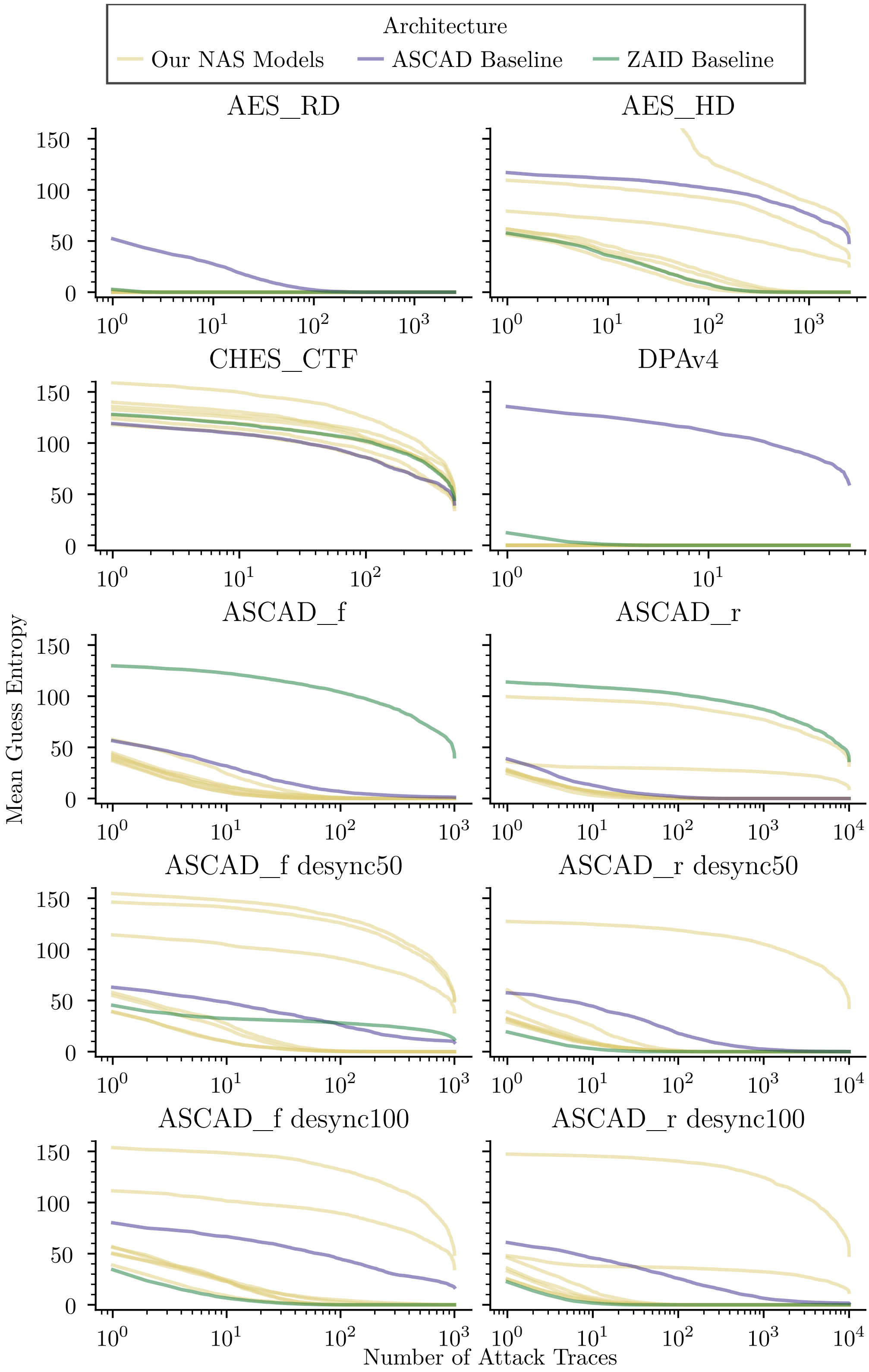


Figure 9: Guess entropy convergence of the 7 NAS models compared to the fixed architecture baseline models for each dataset.

of the 7 models become outliers with significantly slower convergence. One of the models even started with a diverging prediction, only achieving a decent guess entropy towards the very end of the dataset. Similar behavior can be observed with ASCAD_r, where 5 out of 7 models outperform the ASCAD baseline and two outlier models suffer much slower convergence. For the desynchronized datasets, a similar issue occurs, and in total NAS outliers appear to be present in 6 of the 10 datasets we considered. When taking a closer look at the ZAID baseline for ASCAD_f desync50, the baseline itself appears to be affected by a random outlier model, even though it is a fixed architecture.

Generalization We observed the best NAS architecture going from outperforming the state-of-the-art on ASCAD_r and AES_HD to being vastly inferior simply because of a different train-validation dataset split. This indicates that the performance of NAS, like most ML processes, can sometimes vary heavily with small changes and randomization. Considering that the default approach for evaluating ML-based SCAs usually does not repeat the experiment with different train-validation-test splits or other cross-validation techniques, their generalizability is unclear. When looking at CHES_CTF, it also becomes obvious why splitting attack datasets into independent deciles as we did is not done more frequently: The attack dataset appears to be simply too small to be split into 10 parts, as all of the models, although appearing to be converging, simply did not reach a GE of 1 at the end.

When taking all of this into account, we can nonetheless conclude that NAS comes very close to state-of-the-art fixed architectures in terms of attack efficiency, but it is not able to do so consistently. This is promising, as it means that manual per-dataset architecture design can potentially be avoided altogether with NAS.

6 Conclusions and Future Work

In this paper, we proposed to perform a black-box objective (using only profiling dataset) NAS to perform the SCA on hardware systems containing ID and HW leakages. In order to understand the impact of different NAS parameters like the search strategy and input shape of the data (1-D or 2-D) on the performance of the best model, we performed a detailed parameter study. We considered the 4 search strategies implemented by `AutoKeras`, Random, Greedy, Hyperband, and Bayesian. We also considered converting the original one-dimensional inputs to two-dimensional rectangular and square inputs, enabling us to use 2-D CNNs architectures. These choices were combined to create a large-scale parameter study, investigating the influence of search strategy and input shape on 10 different datasets and in the ID and HW leakage models. In order to get a better estimate of the success rate of each combination, we repeat the attack 10 times over different attack dataset parts, for 7 independent NAS models. Upon detailed analysis of the results of the study, we concluded that using the Random search strategy on one-dimensional inputs yields the best-performing CNN architectures for the medium-sized computational budgets we used. The attack performance is overall worse in the HW leakage model, presumably due to the presence of a large imbalance in the dataset [Pic+18]. We also compared the efficiency of these NAS models with previously proposed state-of-the-art CNN baselines. We showed that for most of the synchronized datasets, the 7 models produced by applying NAS were more efficient and required less attack traces than the baseline.

Considering that our approach was able to match the performance of hand-crafted architectures, NAS allows for fully automated attacks on devices or datasets with unknown characteristics. Our experiments highlight the importance of exploration, which needs to be considered when choosing search strategies for hardware attacks in the future. Given that a real-world attacker is likely to commit even bigger budgets to an attack, which enables Random search to find even better architectures, our results should be considered only a lower limit to the capabilities of actual attackers. The apparent ability of NAS to generate useful architectures in various circumstances also allows for non-biased comparison of side-channel attack methods where current comparisons are skewed by the fixed architecture that is considered, e.g. loss functions. A big issue we observed was the susceptibility of the ML models to small variations in the training datasets, which current works applying ML to SCA are not accounting for. We were able to spot this issue because of the repeated training with different training-validation splits, but a broader discussion on how to achieve more consistent performance evaluations when applying ML to SCA needs to take place.

Possible future improvements could mitigate the imbalance in the HW leakage model, which negatively affected our results. This could be addressed by moving away from optimizing for accuracy alone towards more elaborate metrics such as balanced accuracy, Matthew's correlation coefficient, or AUC-score [GBV20]. Additionally, there are different class weighting techniques proposed in the literature which penalize the misclassification of a minority class more than that of a majority class, which could improve the convergence and learning of a CNN [HK18]. Given the good performance observed in our study, it would also be interesting to explore more difficult datasets like ASCADv2, or to use full input traces instead of truncated ones. We constrained our study to four search strategies, but there are more sophisticated alternatives that have been observed to be more time-efficient and more effective than e.g. Random search in finding an optimal architecture [Ren+21]. Another possible efficiency improvement would be early stopping, where some hyperparameter optimization runs are aborted early if the performance is particularly underwhelming or if no further improvement occurs.

7 Acknowledgments

We would like to thank Karlson Pfanschmidt, Alexander Tornede, Tibor Jager, Arunselvan Ramaswamy, Gabriel Zaid, Łukasz Chmielewski, Maikel Kerkhof, Guilherme Perin and Stjepan Picek for their valuable and helpful suggestions. A special thanks goes to Varun Nandkumar Golani, who helped with the experimental setup and ran some of the preliminary experiments for this study as a part of his master thesis. This work is supported by the Bundesministerium für Bildung und Forschung (BMBF) under the project 16KIS1190 (AutoSCA) and funded by European Research Council (ERC)-802823. Experiments were performed on resources provided by the Paderborn Center for Parallel Computing (PC2).

References

[AGF22] Rabin Y. Acharya, Fatemeh Ganji, and Domenic Forte. "Information Theory-based Evolution of Neural Networks for Side-channel Analysis". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.1 (2022), 401–437. doi: 10.46586/tches.v2023.i1.401-437. url: https://tches.iacr.org/index.php/TCHES/article/view/9957.

[Ben+20] Ryad Benadjila et al. "Deep learning for side-channel analysis and introduction to ASCAD database". In: *Journal of Cryptographic Engineering* 10.2 (June 2020), pp. 163–188. doi: 10.1007/s13389-019-00220-8.

[Bha+14] Shivam Bhasin et al. "Analysis and Improvements of the DPA Contest v4 Implementation". In: *Security, Privacy, and Applied Cryptography Engineering - 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*. Ed. by Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont. Vol. 8804. Lecture Notes in Computer Science. Springer, 2014, pp. 201–218. doi: 10.1007/978-3-319-12060-7_14.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. “Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures - Profiling Attacks Without Pre-processing”. In: *CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. Springer, Heidelberg, Sept. 2017, pp. 45–68. DOI: 10.1007/978-3-319-66787-4_3.

[Cha+22] Lipeng Chang et al. “Research on Side-Channel Analysis Based on Deep Learning with Different Sample Data”. In: *Applied Sciences* 12.16 (2022), p. 8246. DOI: 10.3390/app12168246.

[CK09] Jean-Sébastien Coron and Ilya Kizhvatov. “An Efficient Method for Random Delay Generation in Embedded Software”. In: *CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. LNCS. Springer, Heidelberg, Sept. 2009, pp. 156–170. DOI: 10.1007/978-3-642-04138-9_12.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *CHES 2002*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. LNCS. Springer, Heidelberg, Aug. 2003, pp. 13–28. DOI: 10.1007/3-540-36400-5_3.

[Cyb89] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals, and Systems* 2.4 (Dec. 1989), pp. 303–314. DOI: 10.1007/BF02551274.

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Neural Architecture Search: A Survey”. In: *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21. URL: http://jmlr.org/papers/v20/18-598.html.

[FSH15] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. “Initializing Bayesian Hyperparameter Optimization via Meta-Learning”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI’15. Austin, Texas: AAAI Press, 2015, 1128–1135. DOI: 10.5555/2887007.2887164.

[GBV20] Margherita Grandini, Enrico Bagli, and Giorgio Visani. *Metrics for Multi-Class Classification: an Overview*. 2020. DOI: 10.48550/ARXIV.2008.05756. arXiv: 2008.05756.

[GHO15] Richard Gilmore, Neil Hanley, and Maire O’Neill. “Neural network-based attack on a masked implementation of AES”. In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2015, pp. 106–111. DOI: 10.1109/HST.2015.7140247.

[GJS19] Aron Gohr, Sven Jacob, and Werner Schindler. *CHES 2018 Side Channel Contest CTF - Solution of the AES Challenges*. Cryptology ePrint Archive, Report 2019/094. https://eprint.iacr.org/2019/094. 2019.

[Het+20] Benjamin Hettwer et al. “Encoding Power Traces as Images for Efficient Side-Channel Analysis”. In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 46–56. DOI: 10.1109/HOST45689.2020.9300289.

[Heu+20] Annelie Heuser et al. “Lightweight Ciphers and Their Side-Channel Resilience”. In: *IEEE Transactions on Computers* 69.10 (2020), pp. 1434–1448. DOI: 10.1109/TC.2017.2757921.

[HK18] Mahdi Hashemi and Hassan A. Karimi. “Weighted Machine Learning”. In: *Statistics, Optimization & Information Computing* 6.4 (Nov. 2018), pp. 497–525. DOI: 10.19139/soic.v6i4.479.

[Hos+11] Gabriel Hospodar et al. “Machine learning in side-channel analysis: a first study”. In: *Journal of Cryptographic Engineering* 1.4 (Dec. 2011), pp. 293–302. DOI: 10.1007/s13389-011-0023-x.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[Jen00] David Jensen. "Data Snooping, Dredging and Fishing: The Dark Side of Data Mining a SIGKDD99 Panel Report". In: *SIGKDD Explor. Newsl.* 1.2 (Jan. 2000), 52–54. DOI: 10.1145/846183.846195.

[Jin21] Haifeng Jin. "Efficient neural architecture search for automated deep learning". PhD thesis. Texas A&M University, 2021. URL: https://oaktrust.library.tamu.edu/bitstream/handle/1969.1/193093/JIN-DISSERTATION-2021.pdf.

[JSH19] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Ed. by Ankur Teredesai et al. ACM, 2019, pp. 1946–1956. DOI: 10.1145/3292500.3330648.

[Kas+21] Priyank Kashyap et al. "2Deep: Enhancing Side-Channel Attacks on Lattice-Based Key-Exchange via 2-D Deep Learning". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2021), pp. 1217–1229. DOI: 10.1109/TCAD.2020.3038701.

[Ker+22] Maikel Kerkhof et al. "Focus is Key to Success: A Focal Loss Function for Deep Learning-Based Side-Channel Analysis". In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by Josep Balasch and Colin O'Flynn. Cham: Springer International Publishing, 2022, pp. 29–48. DOI: 10.1007/978-3-030-99766-3_2.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Heidelberg, Aug. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[Kle17] Matthew Kleinsmith. *CNNs from different viewpoints*. Feb. 2017. URL: https://medium.com/impactai/cnns-from-different-viewpoints-fab7f52d159c.

[LBM15] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. "A machine learning approach against a masked AES - Reaching the limit of side-channel attacks with a learning model". In: *Journal of Cryptographic Engineering* 5.2 (June 2015), pp. 123–139. DOI: 10.1007/s13389-014-0089-3.

[Ler+15] Liran Lerman et al. "Template Attacks vs. Machine Learning Revisited (and the Curse of Dimensionality in Side-Channel Analysis)". In: *COSADE 2015*. Ed. by Stefan Mangard and Axel Y. Poschmann: vol. 9064. LNCS. Springer, Heidelberg, Apr. 2015, pp. 20–33. DOI: 10.1007/978-3-319-21476-4_2.

[Li+17] Lisha Li et al. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), 6765–6816. DOI: 10.5555/3122009.3242042.

[Lom+14] Victor Lomné et al. "How to Estimate the Success Rate of Higher-Order Side-Channel Attacks". In: *CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. LNCS. Springer, Heidelberg, Sept. 2014, pp. 35–54. DOI: 10.1007/978-3-662-44709-3_3.

[LT20] Liam Li and Ameet Talwalkar. "Random Search and Reproducibility for Neural Architecture Search". In: *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*. Ed. by Ryan P. Adams and Vibhav Gogate. Vol. 115. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 367–377. URL: https://proceedings.mlr.press/v115/li20c.html.

[Mas94] J.L. Massey. "Guessing and entropy". In: *Proceedings of 1994 IEEE International Symposium on Information Theory*. 1994, p. 204. DOI: 10.1109/ISIT.1994.394764.

[MDP19] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. "A Comprehensive Study of Deep Learning for Side-Channel Analysis". In: *IACR TCHES* 2020.1 (2019). https://tches.iacr.org/index.php/TCHES/article/view/8402, pp. 348–375. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i1.348-375.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. "Breaking Cryptographic Implementations Using Deep Learning Techniques". In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Claude Carlet, M. Anwar Hasan, and Vishal Saraswat. Cham: Springer International Publishing, 2016, pp. 3–26. DOI: 10.1007/978-3-319-49445-6_1.

[PCP20] Guilherme Perin, Łukasz Chmielewski, and Stjepan Picek. "Strength in Numbers: Improving Generalization with Ensembles in Machine Learning-based Profiled SCA". In: *IACR TCHES* 2020.4 (2020). https://tches.iacr.org/index.php/TCHES/article/view/8686, pp. 337–364. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i4.337-364.

[PHG17] Stjepan Picek, Annelie Heuser, and Sylvain Guilley. "Template attack versus Bayes classifier". In: *Journal of Cryptographic Engineering* 7.4 (Nov. 2017), pp. 343–351. DOI: 10.1007/s13389-017-0172-7.

[Pic+17] Stjepan Picek et al. "Side-channel analysis and machine learning: A practical perspective". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 4095–4102. DOI: 10.1109/IJCNN.2017.7966373.

[Pic+18] Stjepan Picek et al. "The Curse of Class Imbalance in Side-channel Evaluation". In: *IACR TCHES* 2019.1 (2018). https://tches.iacr.org/index.php/TCHES/article/view/7339, pp. 209–237. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i1.209-237.

[Pic+21] Stjepan Picek et al. *SoK: Deep Learning-based Physical Side-channel Analysis*. Cryptology ePrint Archive, Report 2021/1092. https://eprint.iacr.org/2021/1092. 2021.

[PWP21] Guilherme Perin, Lichao Wu, and Stjepan Picek. *AISY - Deep Learning-based Framework for Side-channel Analysis*. Cryptology ePrint Archive, Report 2021/357. https://eprint.iacr.org/2021/357. 2021.

[Ren+21] Pengzhen Ren et al. "A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions". In: *ACM Computing Surveys (CSUR)* 54.4 (May 2021). DOI: 10.1145/3447582.

[Rij+21] Jorai Rijsdijk et al. "Reinforcement Learning for Hyperparameter Tuning in Deep Learning-based Side-channel Analysis". In: *IACR TCHES* 2021.3 (2021). https://tches.iacr.org/index.php/TCHES/article/view/8989, pp. 677–707. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i3.677-707.

[SAS21] Mehwish Shaikh, Qasim Ali Arain, and Salahuddin Saddar. "Paradigm Shift of Machine Learning to Deep Learning in Side Channel Attacks - A Survey". In: *2021 6th International Multi-Topic ICT Conference (IMTIC)*. 2021, pp. 1–6. DOI: 10.1109/IMTIC53841.2021.9719689.

[SZ15] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. DOI: 10.48550/arXiv.1409.1556.

[TPR13] Adrian Thillard, Emmanuel Prouff, and Thomas Roche. “Success through Confidence: Evaluating the Effectiveness of a Side-Channel Attack”. In: *CHES 2013*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. LNCS. Springer, Heidelberg, Aug. 2013, pp. 21–36. DOI: 10.1007/978-3-642-40349-1_2.

[Wou+20] Lennert Wouters et al. “Revisiting a Methodology for Efficient CNN Architectures in Profiling Attacks”. In: *IACR TCHES* 2020.3 (2020). https://tches.iacr.org/index.php/TCHES/article/view/8586, pp. 147–168. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.147-168.

[WPP20] Lichao Wu, Guilherme Perin, and Stjepan Picek. *I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis*. Cryptology ePrint Archive, Report 2020/1293. https://eprint.iacr.org/2020/1293. 2020.

[Zai+19] Gabriel Zaid et al. “Methodology for Efficient CNN Architectures in Profiling Attacks”. In: *IACR TCHES* 2020.1 (2019). https://tches.iacr.org/index.php/TCHES/article/view/8391, pp. 1–36. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i1.1-36.