Reinforcement Learning for Hyperparameter Tuning in Deep Learning-based Side-channel Analysis

Jorai Rijsdijk[1], Lichao Wu[1], Guilherme Perin[1] and Stjepan Picek[1]

Delft University of Technology, The Netherlands

Abstract. Deep learning represents a powerful set of techniques for profiling side-channel analysis. The results in the last few years show that neural network architectures like multilayer perceptron and convolutional neural networks give strong attack performance where it is possible to break targets protected with various countermeasures. Considering that deep learning techniques commonly have a plethora of hyperparameters to tune, it is clear that such top attack results can come with a high price in preparing the attack. This is especially problematic as the side-channel community commonly uses random search or grid search techniques to look for the best hyperparameters.

In this paper, we propose to use reinforcement learning to tune the convolutional neural network hyperparameters. In our framework, we investigate the Q-Learning paradigm and develop two reward functions that use side-channel metrics. We mount an investigation on three commonly used datasets and two leakage models where the results show that reinforcement learning can find convolutional neural networks exhibiting top performance while having small numbers of trainable parameters. We note that our approach is automated and can be easily adapted to different datasets. Several of our newly developed architectures outperform the current state-of-the-art results. Finally, we make our source code publicly available. [1]

Keywords: Side-channel Analysis · Deep Learning · Reinforcement Learning · Reward · Q-policy · Hyperparameter tuning · Convolutional Neural Networks

1 Introduction

Deep learning-based side-channel analysis (SCA) represents a powerful option for profiling SCA. The results in just a few years showed the potential of such an approach, see, e.g., [MPP16, KPH+19, ZBHV19]. This potential is so significant that most of the SCA community turned away from simpler machine learning techniques, representing the go-to approaches only a few years ago. [2] Intuitively, we can find two main reasons for such popularity of deep learning-based SCA 1) strong performance: breaking targets protected even with countermeasures, 2) no need for pre-processing: beyond selecting some window of features, it seems the researchers are not interested anymore in finding the most informative features. Simultaneously, there are also some issues in using deep learning 1) deep learning models are commonly much more complex than the models obtained with simpler machine learning techniques. As such, a goal like explainable machine learning is more difficult to reach, 2) deep learning architectures are complex, meaning there are many hyperparameters

[1] https://github.com/AISyLab/Reinforcement-Learning-for-SCA

[2] In the last few years, there appears to be only a handful of works investigating profiling SCA while not (exclusively) using deep learning.

one needs to select to find the best-performing one. In this work, we concentrate on the second issue, and more precisely, on how to improve the hyperparameter tuning.

Hyperparameter tuning is an important aspect and by no means unique to the SCA domain. Indeed, without finding strong deep learning models, we cannot expect to reach top attack performance. Besides, if we do not utilize the full power of deep learning, why should we even use such complex techniques? Fortunately, over the years, researchers devised a number of techniques to (efficiently) search for hyperparameters.

Hyperparameter tuning is also a difficult problem. While with simpler machine learning techniques, one could do a handful of experiments and already obtain a good intuition of the performance and the hyperparameter sensitivity, for deep learning, the process is much more complex. Neural networks like convolutional neural networks have many hyperparameters, and exhaustively testing all options is impossible. Simple tuning techniques like random search and grid search can bring good results but depend on luck and the machine learning designer's experience (concerning the specificity of datasets, countermeasures, and leakage models).

There are more powerful options for hyperparameter tuning. Some common options include Bayesian optimization, evolutionary algorithms, and reinforcement learning. While the first two options received (minimal) attention in the SCA community [MPP16, WPP20], reinforcement learning for hyperparameter tuning is still left unexplored in deep learning-based SCA [3].

In this paper, we aim to fill in this gap by proposing the first (to the best of our knowledge) reinforcement learning framework for deep learning-based SCA hyperparameter tuning. We use a well-known paradigm called Q-Learning, and we devise SCA-oriented reward functions. Our analysis includes 1) the goal of finding top-performing convolutional neural networks (CNNs) and 2) CNNs that are small (in terms of trainable parameters) but exhibit strong attack performance. More precisely, our main contributions are:

- We propose the reinforcement learning framework for hyperparameter tuning for deep learning-based SCA. The framework enables automated and powerful search for CNNs for profiling SCA.
- We motivate and develop custom reward functions for hyperparameter tuning in SCA.
- We conduct extensive experimental analysis considering three datasets and two leakage models.
- We report on a number of newly developed CNN architectures that outperform state-of-the-art results.
- Our code is open-source and available at `https://github.com/AISyLab/Reinforcement-Learning-for-SCA`.

2 Background

2.1 Notation

Calligraphic letters ($\mathcal{X}$) denote sets, and the corresponding upper-case letters (X) denote random variables and random vectors $\mathbf{X}$ over $\mathcal{X}$. The corresponding lower-case letters x and $\mathbf{x}$ denote realizations of X and $\mathbf{X}$, respectively. We denote the key candidate as k where $k \in \mathcal{K}$, and k^* denotes the correct key.

We define a dataset as a collection of traces (measurements) $\mathbf{T}$. Each trace $\mathbf{t}_i$ is associated with an input value (plaintext or ciphertext) $\mathbf{d}_i$ and a key $\mathbf{k}_i$. We divide the dataset into three parts: profiling set consisting of N traces, validation set consisting of V traces, and attack set consisting of Q traces. We denote the vector of learnable parameters

[3] We are aware of one work using reinforcement learning in SCA but in a drastically different setup as discussed in Section 3

in our profiling models as $\boldsymbol{\theta}$ and the set of hyperparameters defining the profiling model as $\mathcal{H}$.

2.2 Machine Learning-based SCA

Commonly, when considering block ciphers, we use the supervised machine learning paradigm, and we investigate the multi-class classification task (with c discrete classes). The task is to learn a function f that maps an input to the output ($f : \mathcal{X} \rightarrow Y$)) based on examples of input-output pairs. The function f is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^n$, where n denotes the number of trainable parameters.

Supervised learning consists of two phases: training and test. We assume a setup where the training phase corresponds to the SCA profiling phase, and the testing phase corresponds to the attack phase in SCA. As such, we use the terms profiling/training and attacking/testing interchangeably.

1. The profiling phase aims to learn $\boldsymbol{\theta}'$ that minimizes the empirical risk represented by a loss function L [4] on a profiling set of size N.

2. The goal of the attack phase is to make predictions about the classes

$$y(x_1, k^*), \ldots, y(x_Q, k^*),$$

where k^* represents the secret (unknown) key on the device under the attack. Finally, y denotes the output of a machine learning model when the input is the key k^* and trace x, i.e., for input x_i, we obtain the machine learning prediction y_i.

As common in deep learning-based SCA, the outcome of predicting with a model f on the attack set is a two-dimensional matrix P with dimensions equal to $Q \times c$. Every row of the matrix P is a vector containing all class probabilities for a specific trace $\mathbf{x}_i$ (note that $\sum_v^c \mathbf{p}_{i,v} = 1, \forall i$. The probability $S(k)$ for any key byte candidate k is used as an SCA distinguisher (commonly the maximum log-likelihood distinguisher):

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

The value $\mathbf{p}_{i,v}$ denotes the probability for a certain class v being selected (the class v is calculated for a key k and input d_i through a cryptographic function and a leakage model l.

To prevent overfitting (i.e., the behavior where the developed model does not generalize to the previously unseen data, it is common to evaluate the behavior of the neural network model being developed on a separate validation set of size V.

In SCA, an adversary is not interested in predicting the classes in the attack phase but revealing the secret key k^*. To estimate the effort required the break the target, it is common to use metrics like guessing entropy (GE) [SMY09] and consider varying number of attack traces. More precisely, given Q traces in the attack phase, an attack outputs a key guessing vector $\mathbf{g} = [g_1, g_2, \ldots, g_{|\mathcal{K}|}]$ in decreasing order of probability (g_1 is the most likely key candidate and $g_{|\mathcal{K}|}$ the least likely key candidate). Guessing entropy is the average position of k^* in $\mathbf{g}$.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) commonly consist of three types of layers: convolutional layers, pooling layers, and dense (fully connected) layers. The convolution layer

[4] Commonly, in deep learning-based SCA, one uses the categorical cross-entropy loss function.

computes neurons' output connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. Pooling decrease the number of extracted features by performing a down-sampling operation along the spatial dimensions. The dense layers compute either the hidden activations or the class scores.

2.4 Neural Network Size

We use reinforcement learning to tune the hyperparameters, where one of our goals is to find as small neural networks as possible (measured through the number of trainable parameters). Indeed, smaller networks are faster to tune, and train and (hopefully) easier to understand. Additionally, considering the SCA evaluation in a security lab, smaller networks would also be preferable because of the limited budget and available time.

Hyperparameters are all configuration variables external to the model f, e.g., the number of hidden layers or activation functions in a neural network. The parameter vector $\boldsymbol{\theta}$ are the configuration variables internal to the model f and are estimated from data. The number of trainable parameters depends on the type of neural network and its architecture size (thus, on hyperparameters). To derive the number of trainable parameters for CNN, we start with the dense layer as used in multilayer perceptron (MLP) Let us consider a setup where each perceptron in the multilayer perceptron (MLP) architecture receives more than one input. Each input x is given its weight w, multiplied by that input's value. The sum of the weighted inputs is then calculated, with an additional bias value b. Finally, the activation function A is applied:

$$y = A(\sum_i w_i \cdot x_i + b). \tag{2}$$

Extending this to the whole MLP, the number of trainable parameters equals the sum of connections between layers summed with biases in every layer:

$$n = (in \cdot r + r \cdot out) + (r + out), \tag{3}$$

where in denotes input size, r is the size of hidden layer(s), and out denotes the output size.

For convolutional neural networks, the number of trainable parameters in one convolution layer equals:

$$n = [in \cdot (fi \cdot fi) \cdot out] + out, \tag{4}$$

where in denotes the number of input maps, fi is the filter size, and out is the number of output maps. As CNNs commonly have dense layers, we additionally use Eqs. 2 and 3 to calculate the full number of trainable parameters.

2.5 Datasets and Leakage Models

We investigate two leakage models:

1. The Hamming weight (HW) leakage model: the attacker assumes the leakage proportional to the sensitive variable's Hamming weight. When considering the AES cipher (or, more precisely, any cipher with an 8-bit S-box), this leakage model results in nine classes for a single intermediate byte.

2. The Identity (ID) leakage model: the attacker considers the leakage in the form of an intermediate value of the cipher. When considering the AES cipher (8-bit S-box), this leakage model results in 256 classes for a single intermediate byte.

2.5.1 ASCAD Datasets.

The first dataset we use is the ASCAD database [BPS+20]. It contains the measurements from an 8-bit AVR microcontroller running a masked AES-128 implementation. For both versions, we attack the first masked key byte (key byte 3). We use 5 000 traces from the attack set for validation purposes. These datasets are available at https://github.com/ANSSI-FR/ASCAD.

There are two versions of the ASCAD dataset: one with a fixed key with 50 000 traces for profiling and 10 000 for the attack. The traces in this dataset have 700 features (preselected window). We use 45 000 traces for profiling and 5 000 for validation from the profiling set.

The second version has random keys for profiling, and the dataset consists of 200 000 traces for profiling and 100 000 for the attack. This dataset has traces consisting of 1 400 features (preselected window).

2.5.2 CHES CTF Dataset.

This dataset consists of masked AES-128 encryption running on a 32-bit STM microcontroller. In our experiments, we use 45 000 traces for training. The training set has a fixed key. The attack set has 5 000 traces and uses a different key than the training set. We use 2 500 traces from the attack set for validation. We attack the first key byte as for this dataset, all key bytes are masked. Each trace consists of 2 200 features. This dataset is available at https://chesctf.riscure.com/2018/news.

2.6 Reinforcement Learning

Reinforcement learning attempts to teach an agent how to perform a task by letting the agent experiment and experience the environment, maximizing some reward signal. It differs from supervised machine learning, where the algorithm learns from a set of examples labeled with the correct answers. An advantage of reinforcement learning over supervised machine learning is that the reward signal can be constructed without prior knowledge of the correct course of action, which is especially useful if such a dataset does not exist or is infeasible to obtain. While, at a glance, reinforcement learning might seem similar to unsupervised machine learning, they are decidedly different. Unsupervised machine learning attempts to find some (hidden) structure within a dataset, whereas finding structure in data is not a goal in reinforcement learning [SB18]. Instead, reinforcement learning aims to teach an agent how to perform a task through rewards and experimentation.

In reinforcement learning, there are two main categories of algorithms, value-based algorithms and policy-based algorithms. Value-based algorithms try to approximate or find the value function that assigns state-action pairs a reward value. These reward values can then be used in a policy. Policy-based algorithms, however, directly try to find this optimal policy.

Most reinforcement learning algorithms are centered around estimating value functions, but this is not a strict requirement for reinforcement learning. For example, methods such as genetic algorithms, genetic programming, and simulated annealing can be used for reinforcement learning without ever estimating value functions [SB18]. In this research, we only focus on Q-Learning, which belongs to the value estimation category.

2.6.1 Q-Learning

Q-Learning was first introduced in 1989 by Chris Watkins [Wat89], and it aims not only to learn from the outcome of a set of state-action transitions but to learn from each of them individually. Q-learning is a value-based algorithm, and it tries to estimate $q_*(s, a)$, the reward of taking an action a in a state s under the optimal policy, by iteratively updating its stored q-value estimations using Eq. (5). The most basic form of Q-learning stores these q-value estimations as a simple lookup table and initializes them with some chosen value or method. This form of Q-learning is also called Tabular Q-learning. The algorithm is illustrated in Figure 1, and the function used to update the current q-value mappings based on the received reward is defined as follows:

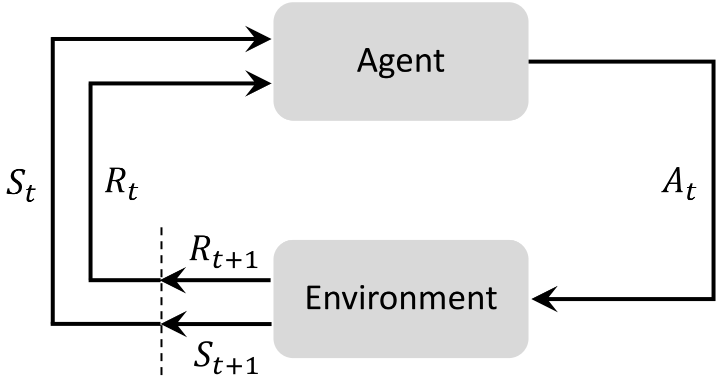


Figure 1: The q-learning concept where an agent chooses an action A_t, based on the current state S_t, which affects the environment. This action is then given a reward R_{t+1} and leads to state S_{t+1}. Eq. (5) is used to incorporate this reward into the saved reward for the current state R_t, and the cycle starts again.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{5}$$

where S_t, A_t are the state and action at time t, and $Q(S_t, A_t)$ is the current expected reward for taking action A_t in state S_t. α and γ are the q-learning rate and discount factor, which are both hyperparameters of the Q-Learning algorithm. The q-learning rate determines how quickly new information is learned, while the discount factor determines how much value we assign to short term versus long term rewards. R_{t+1} is the currently observed reward for having taken action A_t in state S_t. $max_a Q(S_{t+1}, a)$ is the maximum of the expected reward of all the actions a that can be taken in state S_{t+1}.

3 Related Works

Reinforcement learning has been used across a variety of domains and applications. For example, in the domain of bionics, it has been used in classifying human motion intentions from EMG time series with Duelling Deep Q-Learning [SCLW18]. Other applications include using reinforcement learning to counteract class imbalance [LCQ19], including the feature selection as part of the learning process [JPL17, SCLW18], and to select the neural network architectures [BGNR16, ZL16].

Considering hyperparameter tuning in SCA, we can informally divide related works into those that use machine learning and deep learning techniques. Machine learning techniques commonly have much fewer hyperparameters to tune. For example, Naive Bayes [PHG17] has none (the same is true for template attack [CRR02]). For the random forest [Bre01], researchers commonly consider one hyperparameter (the number of trees) [LMBM13]. For SVM [CV95], commonly a few hyperparameters are considered (the kernel type and hyperparameters stemming from that choice. For example, with the most used radial kernel, there are two commonly explored hyperparameters γ and C) [HZ12, PHJ$^+$17]. Additionally, we can consider multilayer perceptron in the machine learning techniques, as many of the first works did not use more than one hidden layer (which would make it deep learning) [GHO15]. Since those techniques have fewer hyperparameters, the hyperparameter tuning was commonly only briefly mentioned or even not discussed at all. In general, we could conclude that hyperparameter tuning did not pose significant challenges for such techniques (it is also possible that we did not use those techniques in the best possible way).

Maghrebi et al. introduced the convolutional neural networks into profiling SCA in 2016 [MPP16]. [5] The authors mentioned they noticed a strong hyperparameter influence on the final results, so they tuned the architectures. While the tuning details are not clear, the authors used genetic algorithms to find the best performing architectures.

Cagli et al. [CDP17] and Picek et al. [PHJ+18] reported very good attack performance even in the presence of countermeasures. Interestingly, the first work did not even discuss hyperparameter tuning, while the second one did the tuning in a manual way (defining a number of possible values for hyperparameters and checking all possible combinations). Kim et al. constructed a VGG-like architecture that performs well over several datasets, but they did not discuss the hyperparameter tuning involved in checking the performance of such an architecture [KPH+19]. Benadjila et al. made an empirical evaluation of different CNN hyperparameters for the ASCAD dataset [BPS+20]. This is one of the first works dealing with the complexity of CNN hyperparameter tuning for SCA. Perin et al. used a random search in pre-defined ranges to build deep learning models to form ensembles [PCP20]. Both of those works reported very good results, despite relatively simple methods to choose hyperparameters.

Next, several works consider the importance of specific hyperparameters in SCA performance. For instance, L. Weissbart investigated multilayer perceptron performance and hyperparameter tuning for the number of layers and neurons, and activation functions [Wei20]. Li et al. investigated the weight initialization for MLP and CNN architectures [LKP20]. Perin and Picek explored the various optimizer choices for deep learning-based SCA [PP20]. We also mention works by, e.g., Zaid et al. [ZBD+21], and Zhang et al. [ZZN+20] where they introduced new loss functions, which extended the range of hyperparameters even more.

Zaid et al. were the first to propose a methodology to select hyperparameters related to the size (the number of learnable parameters: weights and biases) of layers in CNNs. Their approach considers the number of filters, kernel sizes, strides, and the number of neurons in fully-connected layers [ZBHV19]. Wouters et al. [WAGP20] improved upon the work from Zaid et al. [ZBHV19], and discussed several potential problems appearing in the original work. Additionally, Wouters et al. demonstrated how to reach similar attack performance with significantly smaller neural network architectures. Finally, Wu et al. proposed to use Bayesian optimization to find optimal hyperparameters for MLP and CNN architectures [WPP20]. Their results indicated it is possible to find excellent architectures and that even random search can find many architectures that exhibit strong attack performance.

As far as we are aware, there is one paper on reinforcement learning and profiling SCA. There, the authors use reinforcement learning to select LSTM autoencoders for choosing important features [RAD20]. As the authors consider radically different reinforcement learning usage and they attack the ASCON cipher, a comparison between our works is not possible.

We briefly discuss the differences between Bayesian optimization and reinforcement learning approaches. In Bayesian optimization, we consider the black-box approach and define a surrogate function that we optimize instead of the original objective. More precisely, we construct a posterior distribution of functions that best describes the function to be optimized. The next hyperparameter options (values in the search space) are decided

[5] They also used some other deep learning techniques like autoencoders and LSTM, but they are commonly not mentioned due to poor results.

based on the acquisition function results (i.e., the exploration strategy).

Reinforcement learning works in a different setting. There, we have an agent who exists within some environment and can interact with it via a set of actions. The state of the environment changes through time, based on its dynamics and the agent's actions. Additionally, based on the state of the environment and the agent's action, the agent receives a reward. The goal is to have the agent learn a policy (a function that maps from states to actions), maximizing his rewards from the environment. Consequently, the main differences between these two approaches are in the problem setup and the objects involved in each task.

4 The Reinforcement Learning Framework

In this section, we start by discussing the MetaQNN algorithm. Then, we give details on our setup, reward functions, Q-Learning learning rate, and the search space of hyperparameters that we consider.

4.1 MetaQNN

Baker *et al.* introduced MetaQNN, a meta-modeling algorithm, which uses reinforcement learning to automatically generate high-performing CNN architectures in the image classification domain [BGNR17]. The algorithm considers the task of using Q-Learning in training an agent at the task of sequentially choosing neural network layers and their hyperparameters. When reaching a termination state (either a Softmax or global average pooling layer), the MetaQNN algorithm evaluates the generated neural network's performance and, using the accuracy as the reward, uses Q-Learning to adjust the expected reward of the choices made during the neural network generation.

4.2 General Setup

Applying MetaQNN to side-channel analysis is not as simple as simply changing the dataset to side-channel traces and using its accuracy as the reward function. First, conventional machine learning metrics, and especially accuracy, are not a good metric for assessing neural network performance in the SCA domain [PHJ+18]. Second, MetaQNN uses a fixed α (learning rate) for Q-Learning, while using a learning rate schedule where α decreases either linearly or polynomially is the normal practice [EDM04]. Finally, one of the shortcomings of MetaQNN is that it requires either a tremendous amount of time or computing resources to properly explore the neural network search space when we factor in the combination of all the types of layers, their respective parameters, and neural network depths possible. We address this by guiding our search and limiting the search space based on choices motivated by the current state-of-the-art SCA research.

4.3 Reward Functions

To allow MetaQNN to be used for SCA neural network generation, we use a more complicated reward function in place of just using the network's accuracy on the validation dataset. This reward function incorporates the guessing entropy and is composed of four metrics: 1) t': the percentage of traces required to get GE to 0 out of the fixed maximum attack set size; 2) GE'_{10}: the GE value using 10% of the attack traces; 3) GE'_{50}: the GE value using 50% of the attack traces, and 4) the accuracy of the network on the validation set. The formal definition of the first three metrics are expressed in Eqs. (6), (7), and (8).

$$t' = \frac{t_{max} - min(t_{max},\ \overline{Q}_{t_{GE}})}{t_{max}}. \tag{6}$$

$$GE'_{10} = \frac{128 - min(GE_{10}, 128)}{128}. \tag{7}$$

$$GE'_{50} = \frac{128 - min(GE_{50}, 128)}{128}. \tag{8}$$

Note that the first three metrics of the reward function are derived from the GE metric, aiming to reward neural network architectures based on their attack performance using the configured number of traces. Specifically, the second and third metrics are designed for cases that the models require more traces (than the maximum attack traces) to retrieve the secret key. Our reward function in this approach will adequately reward even a model that failed to make GE converge to zero. Furthermore, by including the second and third metrics together in the reward function, the reward function considers the GE convergence, which would better estimate the attack performance of a network. In terms of the fourth metric a (validation accuracy), although related works, e.g., [PHJ+18] indicate a low correlation between validation accuracy and success of an attack, a higher validation accuracy could still mean a lower $\overline{Q}_{t_{GE}}$ (number of traces to reach GE of 0). Therefore, the validation accuracy is added to the reward function. This is especially true if considering the ID leakage model, or reaching a high validation accuracy, meaning that the network classifies correctly. Combining these four metrics, we define the reward function as in Eq. (9), which then gives us the total reward between 0 and 1, since each individual metric is also defined to be a value between 0 and 1. To better reward the model that can retrieve the secret key with fewer traces, larger weights are set on t' and GE'_{10}. We note that the derived reward function is based on significant experimental results lasting for months. Although it is possible to improve the reward function for a specific dataset and a straightforward approach is to tune each metric's weight, a reward function working for different datasets and leakage models better than the one we found should be nontrivial to obtain. Furthermore, we do not claim that the reward function we use is optimal. Rather, we experimentally confirm it gives good results for various experimental settings. Finally, to better assess the contribution of each part of the reward function, we provide additional experiments in Section C.

$$R = \frac{t' + GE'_{10} + 0.5 \times GE'_{50} + 0.5 \times a}{3}. \tag{9}$$

Neural networks with fewer trainable parameters generally take less time and traces to train. To find small but attack-efficient neural networks, we design an additional reward function. Therefore, the reward function shown in Eq. (9) is adapted with a new metric defined in Eq. (10).

$$p' = \frac{max(0,\ p_{max} - p)}{p_{max}}, \tag{10}$$

where p_{max} is defined as a configurable maximum amount of trainable parameters to reward and p is the amount of trainable parameters in the neural network.

Combining Eqs. (9) and 10 gives us a modified reward function R' as denoted in Eq. (11):

$$R' = \frac{t' + GE'_{10} + 0.5 \times GE'_{50} + 0.5 \times a + p'}{4}. \tag{11}$$

To distinguish the reward function used for the experiments, we denote experiments using the small reward function (R') as defined in Eq. (11) as RS experiments. Those denoted as regular experiments or without any indication make use of the reward function (R) as defined in Eq. (9). To understand how these reward functions are incorporated into the main Q-learning algorithm, see Fig. 1 and Eq. (5).

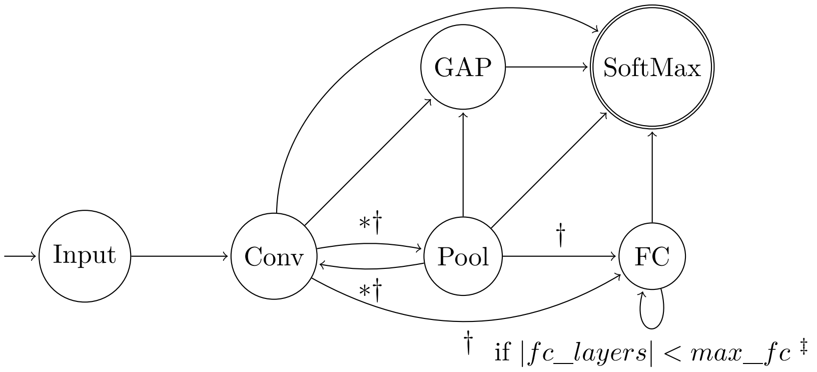


Figure 2: **Original Markov Decision Process for generating a CNN architecture in MetaQNN:** The actions in the process are the addition of specific layers to the neural network being generated. *Only allows transitions to layers with a smaller size than the current representation size. †Only available if the current layer depth is smaller than the maximum configured. ‡max_fc was set to three in Baker *et al.* [BGNR17].

4.4 Q-Learning Learning Rate

Theoretically, Q-Learning converges to the optimal policy with probability one under reasonable conditions on the learning rates and the Markovian environment [WD92]. Furthermore, when using a polynomial learning rate $\alpha = 1/t^{\omega}$ with $\omega \in (1/2, 1)$ and t being the Q-Learning epoch, this convergence is polynomial [EDM04]. Even-Dar *et al.* experimentally found an ω of approximately 0.85 to be optimal across multiple different Markov Decision Processes, which is within their theoretical optimal value range. Therefore, this is also the value we use for all experiments, giving us a learning rate schedule of $\alpha = 1/t^{0.85}$.

4.5 Markov Decision Process Definition and Search Space

The actions in the environment of deciding on neural network layers and their respective parameters are modeled as a Markov Decision Process (MDP) as shown in Figure 3. This MDP differs from the original MetaQNN MDP to search more in the direction of existing state-of-the-art CNNs from the SCA domain, e.g., [ZBHV19]. The original MDP used by Baker *et al.* can be found in Figure 2 [BGNR17] and the version used in our experiments in Figure 3.

The first difference is that we introduce the agent's option to select a Batch Normalization layer between a convolutional and pooling layer, making a network converge faster and more stable by re-centering and re-scaling the inputs [IS15]. Another difference is that while in the original MDP, the agent can choose to transition to a SoftMax or GAP (Global Average Pooling) layer from any of the earlier layer states, we opt for the VGG-like approach as used more commonly in SCA [KPH+19, BPS+20, ZBHV19]. This means that we prefer blocks of Convolutional and Pooling layers, only transitioning to fully-connected layer and SoftMax layer when in a pooling layer or when a transition from a batch normalization layer to a pooling layer is no longer possible due to the current representation size.[6] There is an option to transition from a convolutional layer to a

[6] The representation size is similar to the feature size of a trace. The only difference is that the size no longer directly corresponds to the trace features but rather to the intermediate representation of the trace in the CNN. Therefore, this representation's size is called the representation size, which varies throughout a CNN based on the layer parameters.

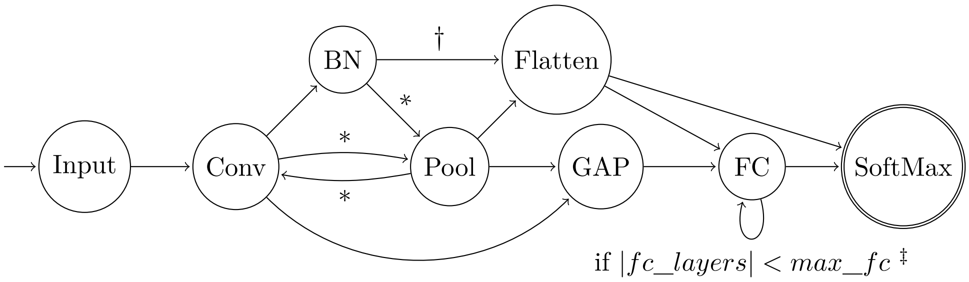


Figure 3: **Our Markov Decision Process for generating a CNN architecture:** The actions in the process are the addition of specific layers to the neural network being generated. *Only allows transitions to layers with a smaller size than the current representation size. †Only available if the transition marked with $*$ is not available due to the current representation size. ‡max_fc was set to three for all experiments, which is consistent with the state-of-the-art SCA results and helps keep the environment size down.

GAP layer as an alternative to a Pool or Flatten layer combination. Another addition is the Flatten layer found in current state-of-the-art SCA CNNs as a transition between convolutional blocks and fully-connected layers. The hyperparameter search space is listed in Table 1. In summary, compared to the original MDP, our new MDP is customized for SCA based on the recent research results. Still, this does not mean it is not possible to use the original MDP or that it would necessarily result in architectures performing poorly. Rather, we consider this as a design choice to limit the search space for reinforcement learning.

Table 1: Hyperparameters for the neural network generation and hyperparameter options for the generated neural networks used for all experiments.

Maximum Total Layers	14
Maximum Fully Connected Layers	3
Fully Connected Layer Size	[2, 4, 10, 15, 20, 30]
Convolutional Padding Type	SAME
Convolutional Layer Depth	[2, 4, 8, 16, 32, 64, 128]
Convolutional Layer Kernel Size	[1, 2, 3, 25, 50, 75, 100]
Convolutional Layer Stride	1
Pooling Layer Filter Size	[2, 4, 7, 25, 50, 75, 100]
Pooling Layer Stride	[2, 4, 7, 25, 50, 75, 100]
SoftMax Initializer	Glorot Uniform
Initializer for other layers	He Uniform
Activation function	SeLU

5 Experimental Results

To assess the SCA performance, we use guessing entropy, where we average 100 independent attacks for guessing entropy calculation. Additionally, we attack a single key byte only (which is properly denoted as the partial guessing entropy), but we denote it as guessing entropy for simplicity.

To assess the performance of the Q-Learning agent, we compare the average rewards per ε [7] and the rolling average of the reward with the expectation, from the principles of Q-Learning, that the average obtained reward increases as the agent improves in selecting neural network architectures suitable for SCA as ε reduces, and the agent starts exploiting.

In terms of computation complexity, eight CPUs and two NVIDIA GTX 1080 Ti graphics processing units (GPUs) (with 11 Gigabytes of GPU memory and 3 584 GPU cores each) are used for each experiment. The memory assigned for each task highly depends on the dataset to be tested. On average, we used 20GB of memory for an experiment. All of the experiments are implemented with the TensorFlow [AAB$^+$15] computing framework and Keras deep learning framework [C$^+$15]. For the time consumption, since more than 2 500 models are examined, four days on average are required to complete the search process. More precisely, we generate 2 700 unique CNNs due to the epsilon-greedy schedule inherited from the MetaQNN paper.

Note that we do not consider multilayer perceptron architectures in this paper despite being commonly used in SCA. We opted for this approach as reinforcement learning is computationally expensive, and results from related works indicate that random search in pre-defined ranges or grid search gives good results for MLP [PHJ$^+$18, PCP20, WPP20]. Furthermore, Bayesian optimization produced top-performing MLP architectures as reported by Wu et al. [WPP20].

Finally, we provide the results for ASCAD with a fixed key and desynchronization equal to 50 in Appendix A and the details on the best-obtained architectures in Appendix B.

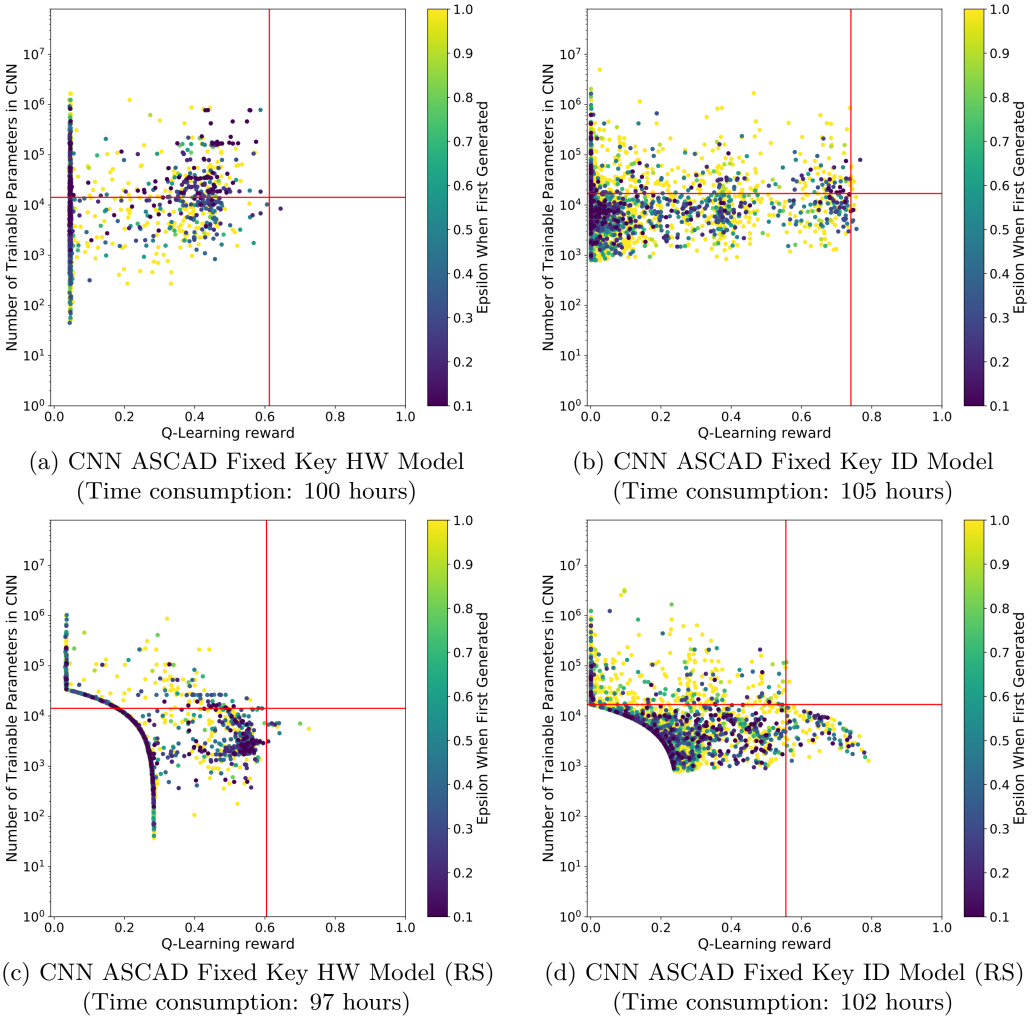
5.1 ASCAD Fixed Key Dataset

Figure 4 depicts the scatter plot results for the HW and ID leakage models, regular and RS reward. For all the experiments, the training batch size is fixed to 50. Notice the red lines that depict the placement of the state-of-the-art model [ZBHV19] concerning the attack performance and the number of trainable parameters. The corresponding reward value is computed by the Q-Learning, using numbers obtained with their publicly available code. The variation of the reward values comes from the usage of different reward equations and leakage models. All dots in the lower right quadrant depict neural network architectures that are smaller and better performing than state-of-the-art. First, we can observe that most of the architectures are larger or worse performing than state-of-the-art. Naturally, this is to be expected as our reinforcement learning framework starts with random architectures. [8] At the same time, for all settings, there are dots in the lower right quadrant, which indicates that we managed to find better performing architectures than in related works. Notice that many architectures are significantly larger when not including the number of trainable parameters in the reward function than those from related work. Interestingly, forcing to find small architectures also results in more highly-fit architectures, suggesting that we do not require large architectures to break this target. Besides, in comparison with random search, the corresponding outcomes are practically equal to yellow dots. As mentioned, the performance of the random search is highly dependent on the pre-defined searching space. As shown in Figure 4, although there are cases where random search obtains good results, the unstable searching performance constrained by other factors makes it a less preferable searching method.

[7] ε schedule for all experiments. A ε of 1.0 means the network was generated completely randomly, while an ε of 0.1 means that the network was generated while choosing random actions 10% of the time.

[8] We start with random architectures to generalize our method's usages in different datasets, allowing the reinforcement learning process to investigate less intuitive hyperparameter combinations. If we start with state-of-the-art architectures, it can easily happen to get stuck there as it would be difficult to find architectures that improve over them.

Notice that we also report the time required to find the neural network models, i.e., for reinforcement learning to finish the process. For this dataset, the time is around 100 hours, slightly more than four days of experiment running (per scenario). It is difficult to give more precise numbers as at the beginning of the process, neural networks vary more in size (and performance), while later in the process, neural networks are more "similar". This discrepancy between the neural networks at the beginning and end of the reinforcement learning process is especially pronounced for the *RS* setting, as final neural networks that are also optimized for size tend to be significantly smaller (thus, faster) than those evaluated at the beginning of the process.



(a) CNN ASCAD Fixed Key HW Model (Time consumption: 100 hours)

(b) CNN ASCAD Fixed Key ID Model (Time consumption: 105 hours)

(c) CNN ASCAD Fixed Key HW Model (RS) (Time consumption: 97 hours)

(d) CNN ASCAD Fixed Key ID Model (RS) (Time consumption: 102 hours)

Figure 4: An overview of the number of trainable parameters, reward, the total search time, and the ε a neural network was first generated for the ASCAD with fixed key dataset experiments. The red lines indicate the amount of trainable parameters and the Q-Learning Reward of the state-of-the-art CNNs from Zaid *et al.*.

Next, in Tables 2 and 3, we provide a comparison of the best-obtained architectures in this paper with results from related works for the HW and ID leakage models, respectively. N/A denotes that the related work does not report on the specific value. The value $\overline{Q}_{t_{GE}}$ denotes the number of attack traces that are required to reach GE of 0. Since [ZBHV19] provides results for the ID leakage model only, the publicly available ID neural network models were adapted to work for the HW leakage model by changing the number of output classes. We note that this approach is not completely "fair" toward the architectures from [ZBHV19], but it represents the best option for the comparison. Notice that for the HW leakage model, we manage to find smaller and better performing architectures than [ZBHV19], regardless of whether we also include the size in the reward function. [WPP20] uses Bayesian optimization and reaches results comparable to our setting while the neural network size is 240 times bigger. The best performing and the smallest network is obtained with reinforcement learning (906 traces to reach GE of 0, and having only 5 566 trainable parameters). We do note that it is not fully fair to compare [WPP20] with regards to the network size, as this is a constraint not considered as a part of their objective function. For the ID leakage model, our results are not so good: Zaid et al. [ZBHV19], Wouters et al. [WAGP20], and Wu et al. [WPP20] reach better performance (especially considering our result where we do not optimize for network size). Still, our best CNN (RS) is significantly smaller than the counterparts, while the performance difference is not so pronounced. Note that [WAGP20] and our best small architecture have similar performance while our network is five times smaller. We believe the reinforcement learning results could be easily improved if taking more human expertise into account. Indeed, as related works indicate very small architectures performing well, we could further constrain the search space size.

Table 2: Comparison of the top generated CNNs for the ASCAD with a fixed key HW leakage model experiments with the current state-of-the-art.

ASCAD Fixed Keys	HW Model			
	[ZBHV19]	[WPP20]	Best CNN	Best CNN (RS)
Trainable Parameters	14 235	1 336 753	8 480	5 566
$\overline{Q}_{t_{GE}}$	1 346	965	1 246	906

Table 3: Comparison of the top generated CNNs for the ASCAD with a fixed key ID leakage model experiments with the current state-of-the-art.

ASCAD Fixed Keys	ID Model				
	[ZBHV19]	[WAGP20]	[WPP20]	Best CNN	Best CNN (RS)
Trainable Parameters	16 960	6 436	3 510 424	79 439	1 282
$\overline{Q}_{t_{GE}}$	191	≈ 200	155	202	242

In Figure 5, we depict the GE results for our best-obtained models for both leakage models and versions of the reward function for ASCAD with the fixed key. There is almost no difference between the two models for the HW leakage model, indicating that reducing the model size did not damage the model performance. The regular reward for the ID leakage model brings somewhat faster GE convergence when considering small attack traces set sizes. The GE difference between Table 3 and Figure 5 comes from the random initialization of the best model's weights before retraining and from the level of detail present in the GE graph, which also applies for the experiments on the other dataset.

In Figure 6, we show the average reward per epsilon and the rolling average of the reward over 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. In this figure, we observe three different behaviors for our Q-Learning agent. As can be seen, when $\varepsilon = 1.0$, the reward value remains relatively flat during the

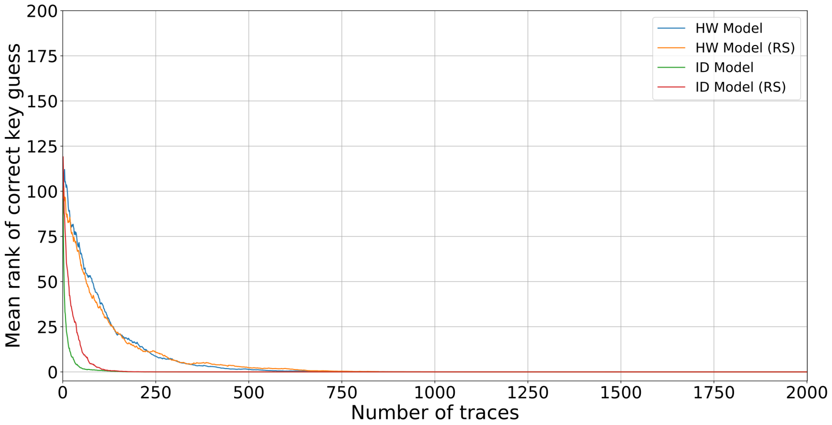


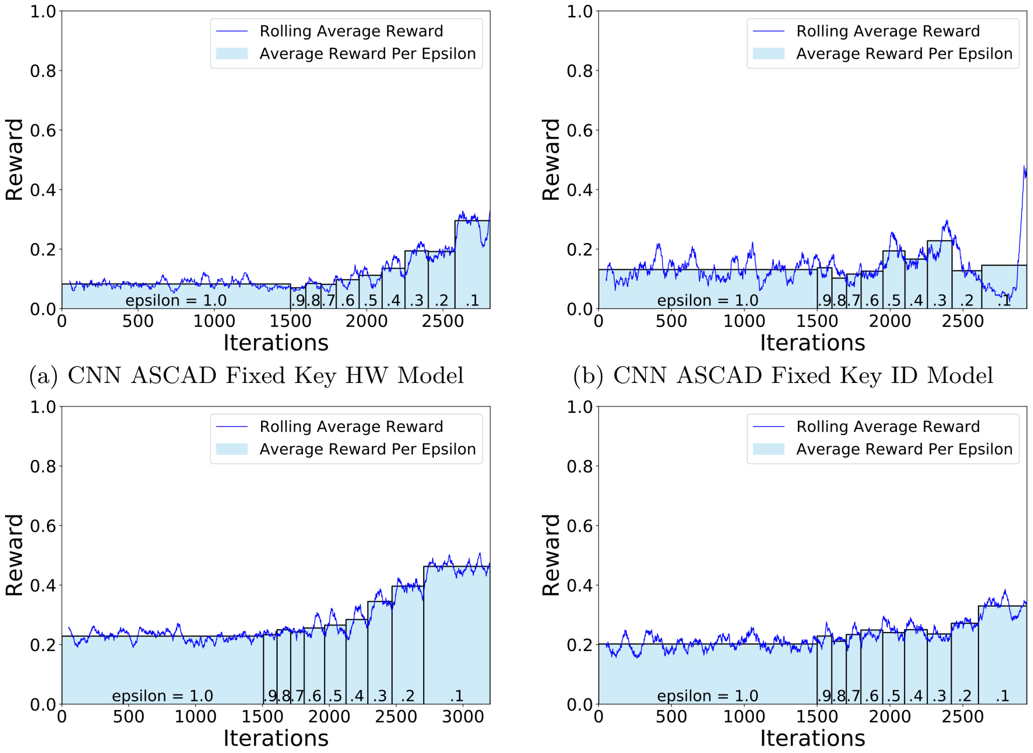
Figure 5: Guessing entropy for the ASCAD with the fixed key dataset.

exploration phase. Indeed, the neural networks are generated randomly when ε equals one. Therefore, this exploration phase can also be viewed as the baseline of the random search. In Figure 6a, the rolling average reward shows a steady increase in the SCA performance of the generated neural network architectures, starting at $\varepsilon = 0.6$. However, this increase is not visible in the average reward across all the generated neural network architectures in each ε until $\varepsilon = 0.1$, where the average reward is approximately 0.41, compared to the average of 0.046 for $\varepsilon = 1.0$. Figure 6b shows the second type of behavior of the Q-Learning agent, where the agent does not seem to show clear signs of increasing the average reward of the neural network architectures it selects as ε decreases. There is a slight upwards trend for ε 0.5 to 0.3, but this does not continue. Fortunately, there is a clear and significant increase in the average reward toward the final iterations of Q-Learning. Finally, we observe the third type of behavior, in Figure 6d, and even more clearly in Figure 6c, where both the average reward per ε and the rolling average reward show a clear and steady increase as ε decreases, indicating that the agent is increasingly able to generate top-performing neural network architectures. It should be noted that the RS experiments have a higher baseline average reward, which occurs due to the added p' component of the R' reward function.

5.2 ASCAD Random Keys Dataset

Figure 7 depicts the results for all the obtained models for the ASCAD with random keys dataset. The training batch size is fixed to 400. We do not depict red lines for this dataset, as we are not aware of results stating precise GE performance and the number of trainable parameters. Still, when we do not optimize the network size, the obtained models' largest grouping is close to zero in terms of Q-Learning reward. If the number of trainable parameters is considered, observe a smooth curve decreasing the number of trainable parameters and increasing the reward. Again, this suggests that while the reward function is more complicated due to an additional term, rewarding smaller models helps find top-performing models. This, in turn, indicates that to find the secret key for this dataset, it is sufficient to use relatively simple neural network architectures, which again means that ASCAD random keys is not much more difficult than ASCAD fixed keys. This is also aligned with the results in [BCH$^+$20], where the authors report more difficulties arising from using different devices than having different keys for training and attack. At the same time, notice slightly larger time consumption that stems from the fact that this dataset is larger compared to ASCAD fixed key.

Tables 4 and 5 give GE and number of trainable parameters comparisons for the HW



(a) CNN ASCAD Fixed Key HW Model

(b) CNN ASCAD Fixed Key ID Model

(c) CNN ASCAD Fixed Key HW Model (RS)

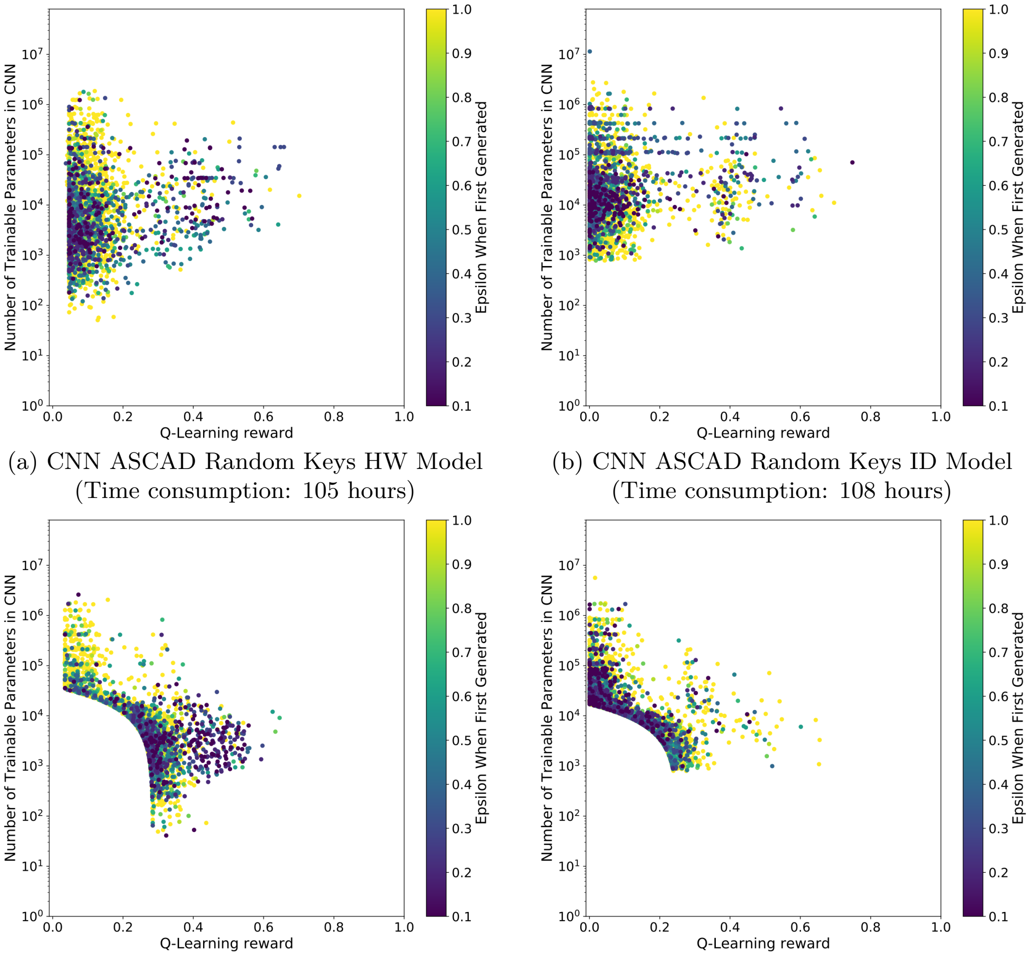
(d) CNN ASCAD Fixed Key ID Model (RS)

Figure 6: An overview of the Q-Learning performance for the ASCAD with fixed key dataset experiments. The black line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. The bars in the graph indicate the average Q-Learning reward for all neural network architectures generated during that ε.

and ID leakage models, respectively. Interestingly, for the HW leakage model, we see that [PCP20] requires significantly fewer traces for GE to reach 0. Still, as that paper considers ensembles of CNNs, a direct comparison is difficult. Also, [WPP20] reaches a similar performance when compared with [PCP20], while the network size is significantly larger than ours. Finally, note that we managed to find a smaller model, but that also comes with a price concerning the GE result. Our result is worse for the ID leakage model than [PCP20], but significantly better than [WPP20]. Interestingly, even the smaller model we found performs much better than the best model found with Bayesian optimization. Again, direct comparison with [PCP20] is not possible because there, the authors use ensembles.

Table 4: Comparison of the top generated CNNs for the ASCAD with random keys HW leakage model experiments with the current state-of-the-art.

ASCAD Random Keys	HW Model			
	[PCP20]	[WPP20]	Best CNN	Best CNN (RS)
Trainable Parameters	N/A	1 314 009	15 241	9 093
$\overline{Q}_{t_{GE}}$	470	496	911	1 264



(a) CNN ASCAD Random Keys HW Model (Time consumption: 105 hours)

(b) CNN ASCAD Random Keys ID Model (Time consumption: 108 hours)

(c) CNN ASCAD Random Keys HW Model (RS) (Time consumption: 102 hours)

(d) CNN ASCAD Random Keys ID Model (RS) (Time consumption: 104 hours)

Figure 7: An overview of the number of trainable parameters, reward, the total search time, and the ε a neural network was first generated for the ASCAD with random key dataset experiments.

Table 5: Comparison of the top generated CNNs for the ASCAD with random keys ID leakage model experiments with the current state-of-the-art.

ASCAD Random Keys	ID Model			
	[PCP20]	[WPP20]	Best CNN	Best CNN (RS)
Trainable Parameters	N/A	2 076 744	70 492	3 298
$\overline{Q}_{t_{GE}}$	105	1 568	490	1 018

Figure 8 gives the GE results for our best-obtained models for both leakage models and versions of the reward function for the ASCAD with random keys dataset. Interestingly, we observe only marginal GE convergence differences for both HW leakage model architectures and the ID leakage model RS architecture. This means that small models can perform well regardless of the leakage model. Still, the architecture for the ID leakage model that uses the regular reward does offer the best performance, especially if the number of traces is smaller than 250.

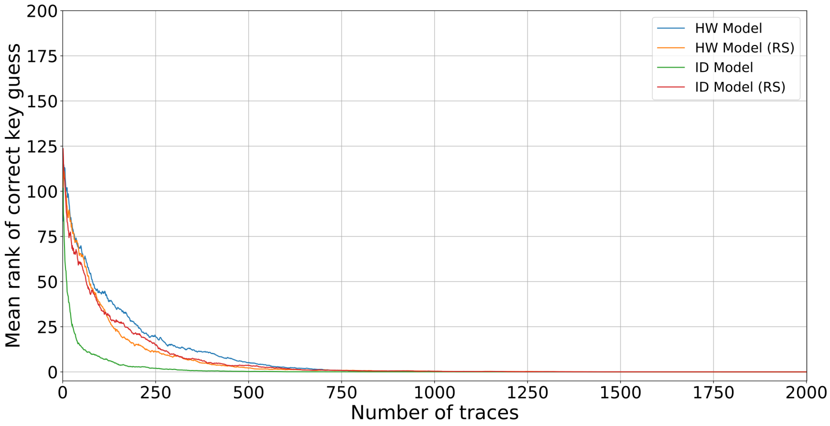


Figure 8: Guessing entropy for the ASCAD with random keys dataset.

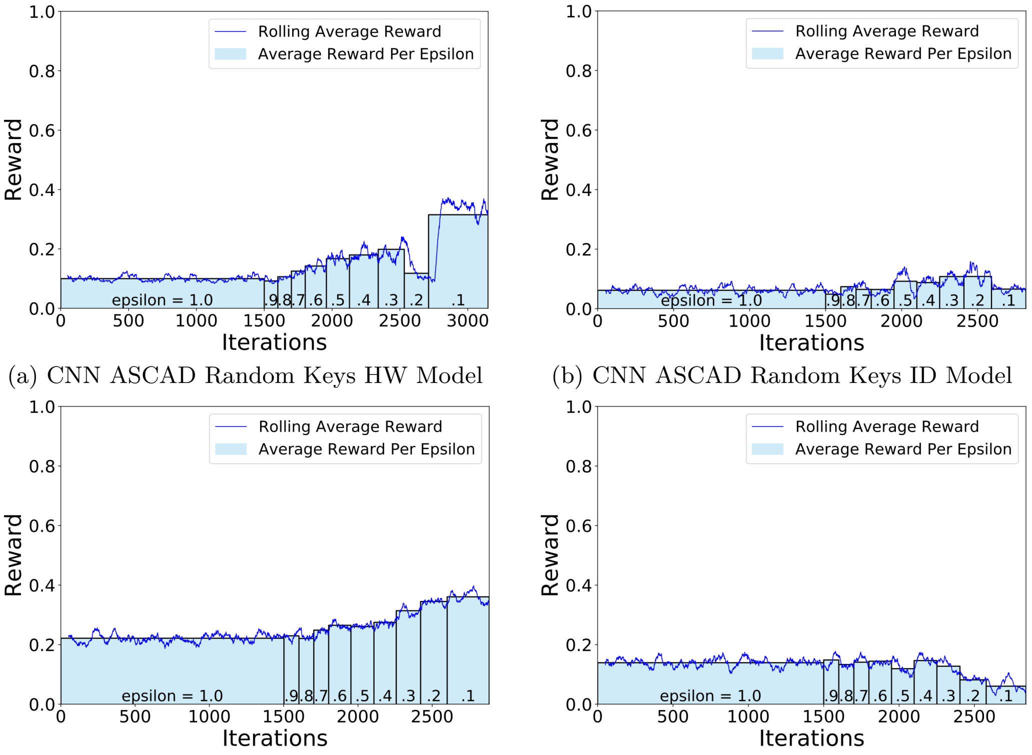
Finally, in Figure 9, we depict the results for the Q-Learning performance for the ASCAD dataset with random keys. The scenarios where we do not reward small sizes are similar to the ASCAD with the fixed key case. There is a steady increase in rolling average reward and the average reward per ε for the HW leakage model, while for the ID leakage model, the average reward slowly increases only after more than 2 000 iterations. For the HW leakage model and RS setting, the results are analogous to the ASCAD fixed key case, where large rolling and average rewards increase with the number of iterations. For the ID leakage model with RS, we observe a new behavior where both rolling and average reward start to decrease after 2 200 iterations. This indicates that reinforcement learning got stuck in local optima, and more iterations only degrade the obtained models' quality.

5.3 CHES CTF Dataset

Finally, we give results for the CHES CTF dataset. We present the HW leakage model results only, as we were unable to find good-performing models in the ID leakage model (also discussed in related works, see, e.g., [PCP20]). The training batch size is set to 400 for all of the following experiments. In Figure 10, we show results for the HW leakage model for the CHES CTF dataset. As before, we do not show red lines as there are no known results that also indicate the number of trainable parameters. Notice that when using the regular reward, most of the models reach a small final reward, while when using a reward with RS, there is a clear tendency toward smaller and better performing models.

Table 6 gives the best obtained results as well as two from related works [PCP20, WPP20]. We cannot compare the number of trainable parameters as related works do not state that information, but we see that our models reach GE with significantly fewer attack traces. Notice that even when we reward smaller models, our attack performance is better than those in related works, and we use a very small architecture. The time consumption for the reinforcement learning process is in line with the previous results, indicating somewhat more than four days of experiments required to finish.

Figure 11 gives the GE results for our best-obtained models for the HW leakage model and both versions of the reward function for the CHES CTF dataset. Observe that the model with the regular reward function performs better when the number of traces is limited, which is in line with the previous results.



(a) CNN ASCAD Random Keys HW Model

(b) CNN ASCAD Random Keys ID Model

(c) CNN ASCAD Random Keys HW Model (RS)

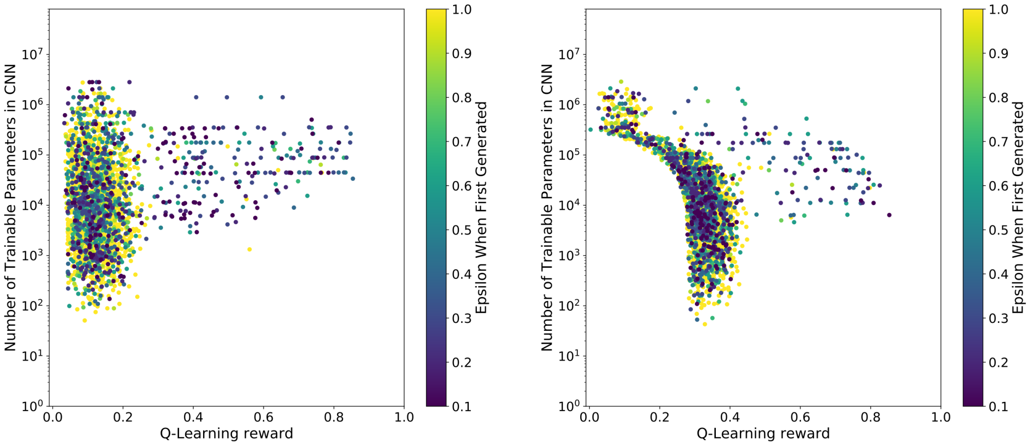
(d) CNN ASCAD Random Keys ID Model (RS)

Figure 9: An overview of the Q-Learning performance for the ASCAD with random key dataset Experiments. The blue line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. The bars in the graph indicate the average Q-Learning reward for all neural network architectures generated during that ε.

Table 6: Comparison of the top generated CNNs for the CHES CTF and HW leakage model experiments with the current state-of-the-art.

CHES CTF	HW Model			
	[PCP20]	[WPP20]	Best CNN	Best CNN (RS)
Trainable Parameters	N/A	2 418 085	33 788	6 395
$\overline{Q}_{t_{GE}}$	310	618	122	349

Finally, in Figure 12, we give results for the Q-Learning performance. Both graphs show a steady increase in the rolling and average rewards as the number of iteration increase. This confirms that our models learn the data and converge to top-performing and small models, as shown in Table 6. Note that our best models are significantly smaller than [WPP20] and perform better.



(a) CNN CHES CTF HW Model (Time consumption: 102 hours)

(b) CNN CHES CTF HW Model (RS) (Time consumption: 108 hours)

Figure 10: An overview of the number of trainable parameters, reward, the total search time, and the ε a neural network was first generated for the CHES CTF dataset experiments.

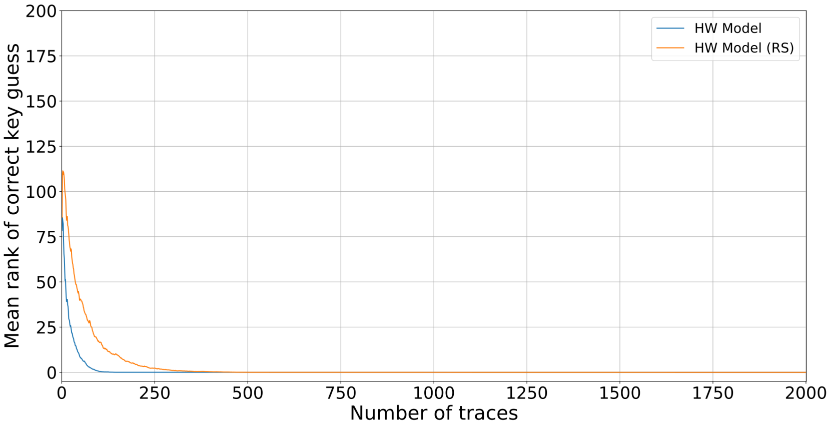
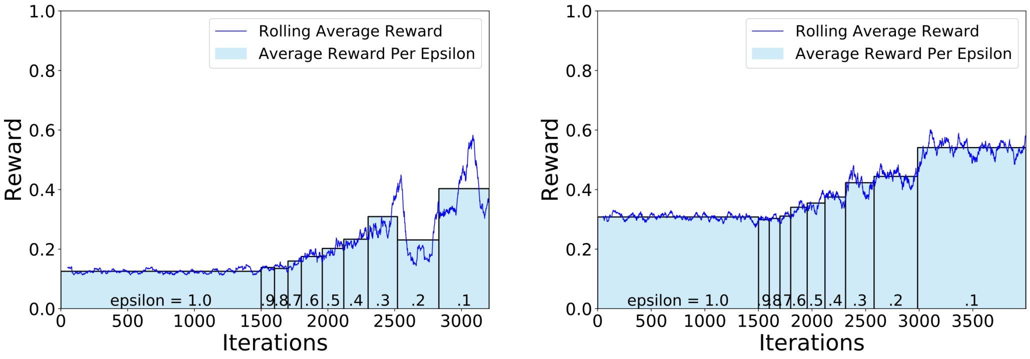


Figure 11: Guessing entropy for the CHES CTF dataset.

6 Conclusions and Future Work

In this paper, we proposed a reinforcement learning framework for deep learning-based SCA. To accomplish that goal, we use a well-known paradigm called Q-Learning, and we define two versions of reward functions that are custom developed for SCA. Additionally, we devise a Markov Decision Process that has a large search space of possible convolutional neural network architectures, but at the same time, still constraints the search following the current state-of-the-art practices in SCA. We test the reinforcement learning behavior for CNN hyperparameter tuning on three datasets and a number of experimental settings. The results show strong performance where we reach the best-known performance in several scenarios, while in other settings, our performance is only moderately worse than state-of-the-art, but our neural network models are extremely small. Additionally, the results we obtained suggest that we should limit the future hyperparameter tuning phases even more as small neural networks often also resulted in the best attack performance. This is especially pronounced for the HW leakage model as smaller networks did not have any performance drawbacks over larger ones. We note that our approach is automated



(a) CNN CHES CTF HW Model

(b) CNN CHES CTF HW Model (RS)

Figure 12: An overview of the Q-Learning performance for the CHES CTF dataset Experiments. The blue line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. The bars in the graph indicate the average Q-Learning reward for all neural network architectures generated during that ε.

and can be easily adapted to different datasets or experimental settings.

There are two research directions we consider particularly interesting for future work. We considered the Q-Learning approach in this work, but there are more powerful (and computationally demanding) approaches available. For instance, it would be interesting to investigate the deep Q-Learning paradigm's performance, especially in a trade-off between computational efficiency and the obtained results. Next, reinforcement learning uses a large number of models before finding the best ones. It would be interesting to consider how well the best models obtained through reinforcement learning would behave in ensembles of models [PCP20]. Additionally, we considered only CNN architectures as we believe their hyperparameter tuning complexity fits into the high computational complexity of reinforcement learning. Still, there are no reasons not to try reinforcement learning with other neural networks, like multilayer perceptrons.

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[BCH+20] Shivam Bhasin, Anupam Chattopadhyay, Annelie Heuser, Dirmanto Jap, Stjepan Picek, and Ritu Ranjan Shrivastwa. Mind the portability: A warriors guide through realistic profiled side-channel analysis. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[BGNR16] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167, 2016.

[BGNR17] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Toulon, France, 2017. International Conference on Learning Representations, ICLR.

[BPS+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *J. Cryptographic Engineering*, 10(2):163–188, 2020.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[C+15] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 45–68, Cham, 2017. Springer International Publishing.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[CV95] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[EDM04] Eyal Even-Dar and Yishay Mansour. Learning rates for q-learning. *J. Mach. Learn. Res.*, 5:1–25, December 2004.

[GHO15] R. Gilmore, N. Hanley, and M. O'Neill. Neural network based attack on a masked implementation of AES. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111, May 2015.

[HZ12] Annelie Heuser and Michael Zohner. Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines. In Werner Schindler and Sorin A. Huss, editors, *COSADE*, volume 7275 of *LNCS*, pages 249–264. Springer, 2012.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Francis Bach and David Blei, editors, *Proceedings of Machine Learning Research*, volume 37, pages 448–456, Lille, France, 2015. PMLR.

[JPL17] Jaromír Janisch, Tomás Pevný, and Viliam Lisý. Classification with costly features using deep reinforcement learning. *CoRR*, abs/1711.07364, 2017.

[KPH+19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 148–179, 2019.

[LCQ19] Enlu Lin, Qiong Chen, and Xiaoming Qi. Deep reinforcement learning for imbalanced classification. *CoRR*, abs/1901.01379, 2019.

[LKP20] Huimin Li, Marina Krček, and Guilherme Perin. A comparison of weight initializers in deep learning-based side-channel analysis. In Jianying Zhou, Mauro Conti, Chuadhry Mujeeb Ahmed, Man Ho Au, Lejla Batina, Zhou Li, Jingqiang Lin, Eleonora Losiouk, Bo Luo, Suryadipta Majumdar, Weizhi Meng, Martín Ochoa, Stjepan Picek, Georgios Portokalidis, Cong Wang, and Kehuan Zhang, editors, *Applied Cryptography and Network Security Workshops*, pages 126–143, Cham, 2020. Springer International Publishing.

[LMBM13] Liran Lerman, Stephane Fernandes Medeiros, Gianluca Bontempi, and Olivier Markowitch. A Machine Learning Approach Against a Masked AES. In *CARDIS*, Lecture Notes in Computer Science. Springer, November 2013. Berlin, Germany.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 3–26. Springer, 2016.

[PCP20] Guilherme Perin, Lukasz Chmielewski, and Stjepan Picek. Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):337–364, Aug. 2020.

[PHG17] Stjepan Picek, Annelie Heuser, and Sylvain Guilley. Template attack versus bayes classifier. *J. Cryptogr. Eng.*, 7(4):343–351, 2017.

[PHJ$^+$17] Stjepan Picek, Annelie Heuser, Alan Jovic, Simone A. Ludwig, Sylvain Guilley, Domagoj Jakobovic, and Nele Mentens. Side-channel analysis and machine learning: A practical perspective. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*, pages 4095–4102, 2017.

[PHJ$^+$18] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):209–237, Nov. 2018.

[PP20] Guilherme Perin and Stjepan Picek. On the influence of optimizers in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/977, 2020. https://eprint.iacr.org/2020/977.

[RAD20] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. Scarl: Side-channel analysis with reinforcement learning on the ascon authenticated cipher, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

[SCLW18] C. Song, C. Chen, Y. Li, and X. Wu. Deep reinforcement learning apply in electromyography data classification. In *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS)*, pages 505–510, 2018.

[Smi17] Leslie N Smith. Cyclical Learning Rates for Training Neural Networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, mar 2017.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 443–461, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[WAGP20] Lennert Wouters, Victor Arribas, Benedikt Gierlichs, and Bart Preneel. Revisiting a methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):147–168, Jun. 2020.

[Wat89] Christopher John Cornish Hellaby. Watkins. *Learning from delayed rewards.* Phd thesis, University of Cambridge England, 1989.

[WD92] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, may 1992.

[Wei20] Leo Weissbart. Performance analysis of multilayer perceptron in profiling side-channel analysis. In Jianying Zhou, Mauro Conti, Chuadhry Mujeeb Ahmed, Man Ho Au, Lejla Batina, Zhou Li, Jingqiang Lin, Eleonora Losiouk, Bo Luo, Suryadipta Majumdar, Weizhi Meng, Martín Ochoa, Stjepan Picek, Georgios Portokalidis, Cong Wang, and Kehuan Zhang, editors, *Applied Cryptography and Network Security Workshops - ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19-22, 2020, Proceedings*, volume 12418 of *Lecture Notes in Computer Science*, pages 198–216. Springer, 2020.

[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, 2020. https://eprint.iacr.org/2020/1293.

[ZBD$^+$21] Gabriel Zaid, Lilian Bossuet, François Dassance, Amaury Habrard, and Alexandre Venelli. Ranking loss: Maximizing the success rate in deep learning side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):25–55, 2021.

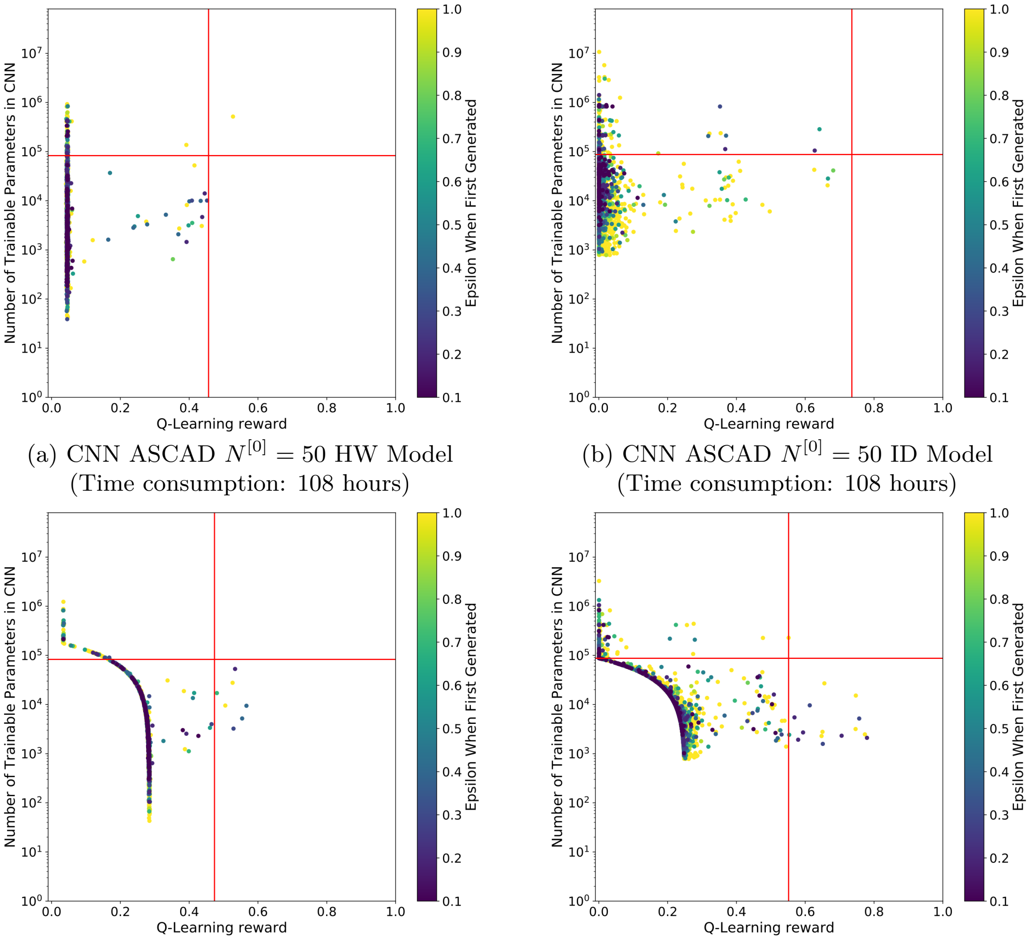
[ZBHV19] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):1–36, Nov. 2019.

[ZL16] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[ZZN$^+$20] Jiajia Zhang, Mengce Zheng, Jiehui Nan, Honggang Hu, and Nenghai Yu. A novel evaluation metric for deep learning-based side channel analysis and its extended application to imbalanced data. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):73–96, Jun. 2020.

A ASCAD Fixed Key with Desynchronization Equal to 50

In Figure 13, we give the results for the ASCAD dataset with the fixed key when there is also desynchronization added. First, observe that this setting poses much more significant problems for reinforcement learning compared to previous ones. Indeed, we do not find CNNs that are better performing and smaller than state-of-the-art for the HW leakage model. A similar observation is also valid for the ID leakage model. Fortunately, when adding the small network constrain to the reward function, we observe much better results, especially for the ID leakage model. We can conclude that desynchronization makes the attack much more difficult, but there are (albeit not many) models that are small and perform well. Constraining the search toward smaller models does improve the search behavior and the final-obtained models' attack performance.



(a) CNN ASCAD $N^{[0]} = 50$ HW Model (Time consumption: 108 hours)

(b) CNN ASCAD $N^{[0]} = 50$ ID Model (Time consumption: 108 hours)

(c) CNN ASCAD $N^{[0]} = 50$ HW Model (RS) (Time consumption: 107 hours)

(d) CNN ASCAD $N^{[0]} = 50$ ID Model (RS) (Time consumption: 103 hours)

Figure 13: **ASCAD $N^{[0]} = 50$:** An overview of the number of trainable parameters, reward, and the ε a CNN was first generated for the ASCAD $N^{[0]} = 50$ experiments. The red lines indicate the amount of trainable parameters and the Q-Learning Reward of the state-of-the-art CNNs from Zaid *et al.*

In Table 7, we observe that our best model is significantly better in terms of GE than [ZBHV19], even though it requires six times more trainable parameters. Interestingly, for our small model, the performance is not much different than [ZBHV19], but we require almost nine times fewer trainable parameters. Table 8 shows the comparison between the best-generated CNNs, including the state-of-the-art reference CNN, and shows that the top-performing value model CNN from the regular experiment does not manage to outperform the state-of-the-art CNN in terms of $\overline{Q}_{t_{GE}}$, requiring almost double the number of traces. However, this CNN does have 0.47 times the number of trainable parameters. The best value model CNN from the RS experiment, on the other hand, needs 69 more traces to retrieve the key, but it only has 0.024 times the number of trainable parameters when compared to the state-of-the-art.

Figure 14 gives the GE results for our best-obtained models for the HW and ID leakage models and both versions of the reward function for the ASCAD with desynchronization equal to 50. In line with previous results, the regular reward model performs better when the number of traces is limited. Figure 15 shows the Q-Learning performance for these experiments where both the rolling average reward of 50 iterations and the average reward per ε stays fairly constant as ε decreases, with Figure 15d showing a decrease for $\varepsilon = 0.1$, which would indicate the agent is stuck in a local optimum. This lack of improvement in average rewards can be expected when the number of top-performing networks is as low as can be observed in Figure 13. We note that some of the resulting best neural networks still outperform the state-of-the-art models.

Table 7: Comparison of the top generated CNNs for the ASCAD $N^{[0]} = 50$ HW model experiments with the current state-of-the-art.

$N^{[0]} = 50$	HW Model		
	[ZBHV19]	Best CNN	Best CNN (RS)
Trainable Parameters	82 879	516 361	9 433
$\overline{Q}_{t_{GE}}$	$> 2\,000$ $GE_{2\,000} = 0.03$	1 592	$> 2\,000$ $GE_{2\,000} = 0.67$

Table 8: Comparison of the top generated CNNs for the ASCAD $N^{[0]} = 50$ ID model experiments with the current state-of-the-art.

$N^{[0]} = 50$	ID Model			
	[ZBHV19]	[WAGP20]	Best CNN	Best CNN (RS)
Trainable Parameters	87 279	41 052	41 321	2 100
$\overline{Q}_{t_{GE}}$	244	≈ 250	443	313

B The Best Obtained Architectures

In Table 1, we present the common hyperparameters that the best-developed architectures use. These are meant to serve as guidelines for future neural network development.

We present the best obtained architectures in Tables 10 to 13. Note that all the tables use the following notation:
Convolutional = C(*filters*, *kernel_size*, *strides*)
Batch Normalization = BN
Average Pooling = P(*size*, *stride*)
Flatten = FLAT
Global Average Pooling = GAP
Fully-connected = FC(*size*)
SoftMax = SM(*classes*)

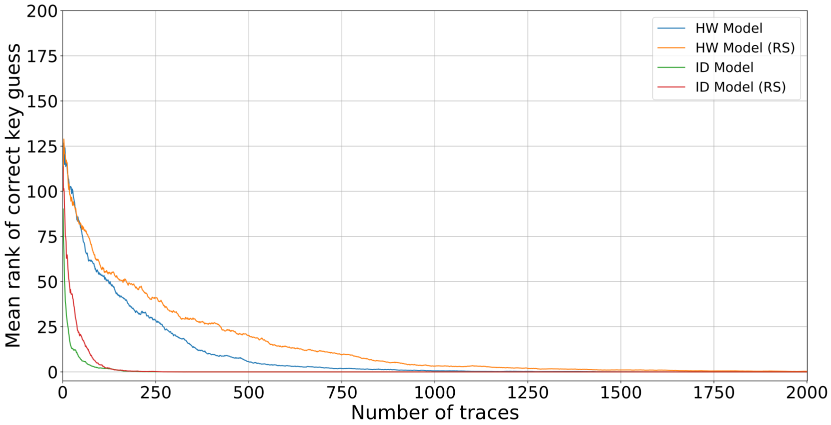
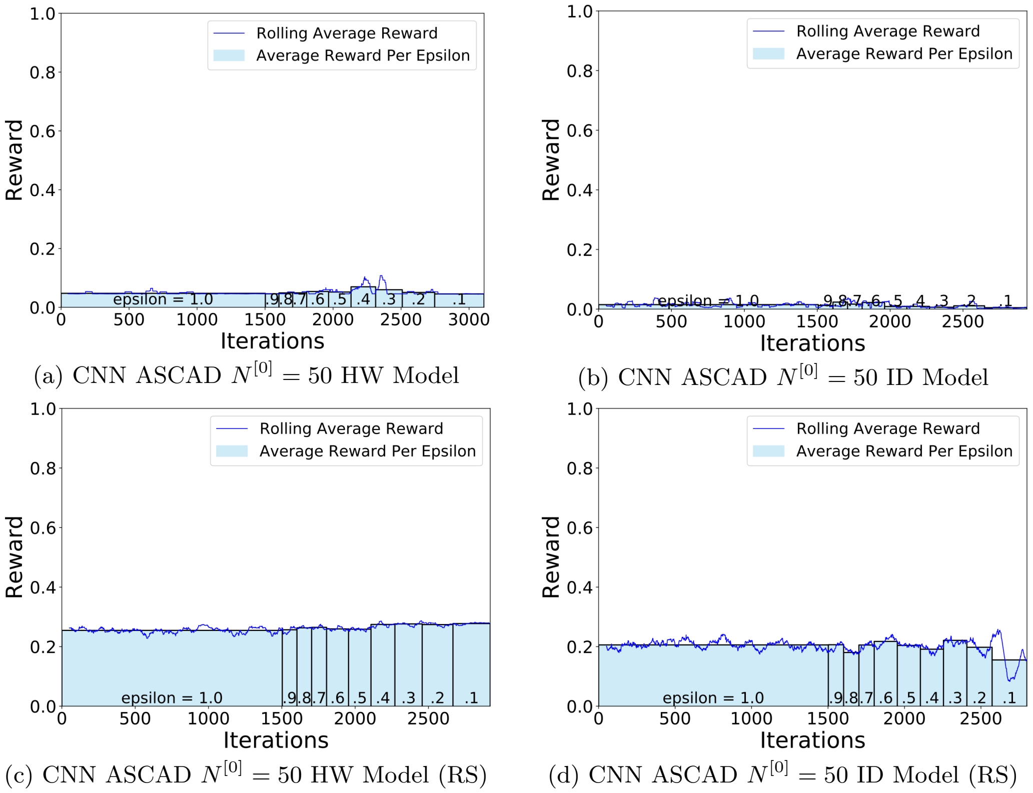


Figure 14: Guessing entropy for the ASCAD fixed key dataset with desynchronization of 50.



(a) CNN ASCAD $N^{[0]} = 50$ HW Model

(b) CNN ASCAD $N^{[0]} = 50$ ID Model

(c) CNN ASCAD $N^{[0]} = 50$ HW Model (RS)

(d) CNN ASCAD $N^{[0]} = 50$ ID Model (RS)

Figure 15: An overview of the Q-Learning performance for the ASCAD fixed key dataset with desynchronization of 50 experiments. The blue line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. The bars in the graph indicate the average Q-Learning reward for all neural network architectures generated during that ε.

Table 9: Common hyperparameters for all the reported best architectures.

Convolutional Padding Type	SAME
Pooling Type	Average Pooling
SoftMax Initializer	Glorot Uniform
Initializer for other layers	He Uniform
Activation function	SeLU
Optimizer	Adam
Train Epochs	50
Learning Rate	One Cycle Policy[Smi17]

B.1 ASCAD Fixed Key

In Table 10, we describe the best-obtained architectures for the ASCAD with the fixed key dataset.

Table 10: The best found architectures for the ASCAD fixed key dataset.

Leakage Model	Type	Architecture
ID	CNN	C(128,25,1), P(25,25), FLAT, FC(20), FC(15), SM(256)
	CNN (RS)	C(2,75,1), P(25,25), C(2,3,1), BN, P(4,4), C(8,2,1), P(2,2), FLAT, FC(10), FC(4), FC(2), SM(256)
HW	CNN	C(16,100,1), P(25,25), FLAT, FC(15), FC(4), FC(4), SM(9)
	CNN (RS)	C(2,25,1), P(4,4), FLAT, FC(15), FC(10), FC(4), SM(9)

B.2 ASCAD Fixed Key with Desynchronization 50

In Table 11, we describe the best-obtained architectures for the ASCAD with the fixed key dataset and desynchronization.

Table 11: The best found architectures for the ASCAD fixed key dataset with desynchronization of 50.

Leakage Model	Type	Architecture
ID	CNN	C(2,50,1), BN, P(50,2), C(16,3,1), P(25,7), C(64,25,1), P(7,7), C(64,3,1), BN, P(4,4), GAP, FC(20), FC(15), FC(4), SM(256)
	CNN (RS)	C(4,50,1), P(50,4), C(2,50,1), P(2,2), C(2,50,1), P(50,50), GAP, FC(4), SM(256)
HW	CNN	C(128,1,1), BN, P(50,4), C(128,25,1), BN, P(25,4), C(32,25,1), P(4,4), C(32,3,1), P(4,4), C(4,1,1), BN, FLAT, FC(10), SM(9)
	CNN (RS)	C(8,3,1), P(50,4), C(32,25,1), P(50,50), FLAT, FC(20), FC(20), FC(20), SM(9)

B.3 ASCAD Random Keys

In Table 12, we describe the best-obtained architectures for the ASCAD with random keys dataset.

Table 12: The best found architectures for the ASCAD random keys dataset.

Leakage Model	Type	Architecture
ID	CNN	C(128,3,1), P(75,75), FLAT, FC(30), FC(2), SM(256)
	CNN (RS)	C(4,1,1), P(100,75), FLAT, FC(30), FC(10), FC(2), SM(256)
HW	CNN	C(8,3,1), P(25,25), FLAT, FC(30), FC(30), FC(20), SM(9)
	CNN (RS)	C(4,50,1), P(25,25), FLAT, FC(30), FC(30), FC(30), SM(9)

B.4 CHES CTF

Finally, in Table 13, we describe the best-obtained architectures for the CHES CTF dataset.

Table 13: The best found architectures for the CHES CTF dataset.

Leakage Model	Type	Architecture
HW	CNN	C(4,100,1), P(4,4), FLAT, FC(15), FC(10), FC(10), SM(9)
	CNN (RS)	C(2,2,1), P(7,7), FLAT, FC(10), SM(9)

C Q-Learning Reward Function

To assess our Q-Learning reward function's performance, we perform additional experiments using the ASCAD fixed key dataset for the HW leakage model. Note that we do not look at the network size component, which has already been covered in the paper's main body. We conduct our experiments using the same parameters as can be found in Section 5. The only variation between these experiments is in the definition of the reward function. Additionally, we do not give results for other datasets, but we note that the results are in line with the results for the ASCAD with the fixed key dataset and the HW leakage model.

C.1 Validation Accuracy

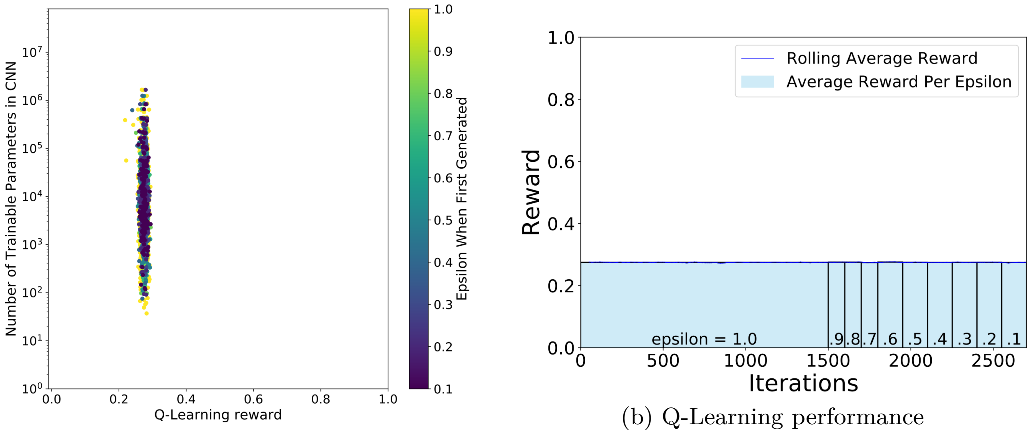
The first component of the reward function we examine is the validation accuracy a. For this $R = a$ experiment, we observe from Figure 16 that the accuracy on its own does very little in terms of distinguishing between the different generated CNNs. On its own, the validation accuracy does not appear suitable as a reward function for our purpose, further solidifying our choice to define a new Q-Learning reward function for the SCA domain.

C.2 GE'_{10}

For the $R = GE'_{10}$ experiment, we observe from Figure 17 that GE'_{10} does properly distinguish between CNNs with different SCA performance. This is also true for the top performing CNNs, as none of them achieve $GE = 0.0$ within 10% of the configured maximum number of traces. However, the Q-Learning performance does not show a significant improvement over randomly selected CNNs as ε decreases, making GE'_{10} unsuitable to use on its own.

C.3 GE'_{50}

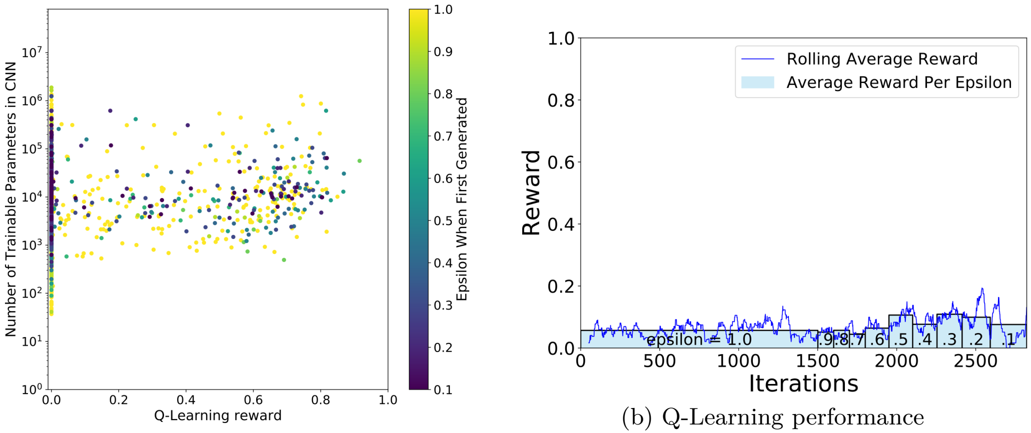
For the $R = GE'_{50}$ experiment, we observe from Figure 18 that while the Q-Learning performance graph shows that the agent behaves as expected with the average reward



(a) Scatter plot of generated networks

(b) Q-Learning performance

Figure 16: An overview of both the number of trainable parameters, reward, ε, and Q-Learning performance when $R = a$.



(a) Scatter plot of generated networks

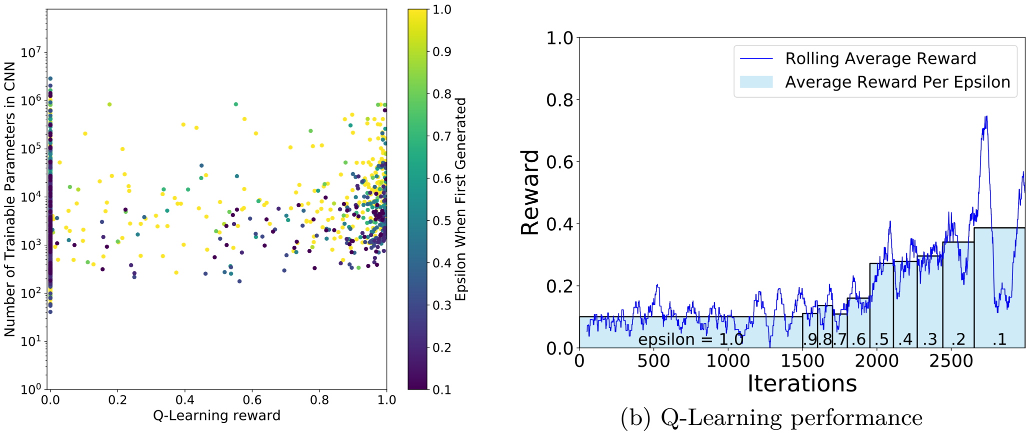
(b) Q-Learning performance

Figure 17: An overview of both the number of trainable parameters, reward, ε, and Q-Learning performance when $R = GE'_{10}$.

increasing as the epsilon decreases, the scatter plot shows that a significant number of networks manage to achieve a GE_{50} of 0, where CNNs manage to retrieve the key with fewer than 50% of the configured traces. This highlights the main weakness of using this reward component on its own.

C.4 t'

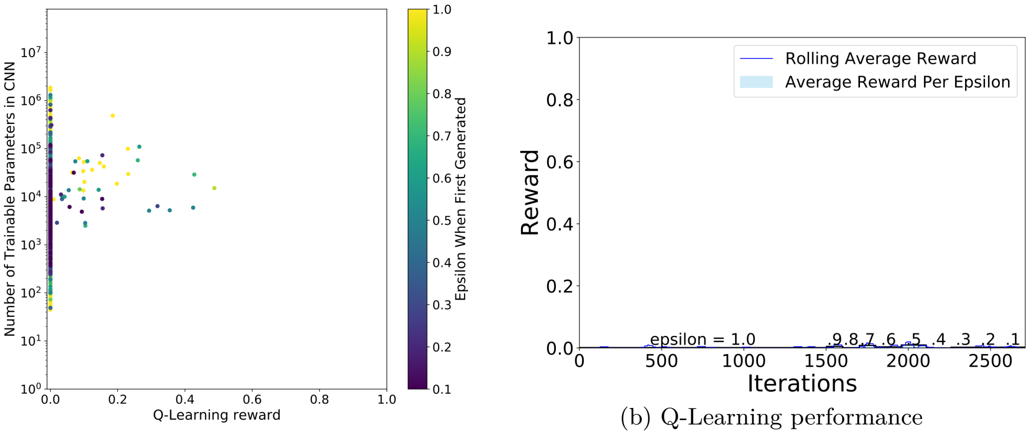
For the $R = t'$ experiment, we observe in Figure 19 that only very few generated networks actually manage to retrieve the secret key within the configured number of traces, making it hard for the Q-Learning agent to learn enough to start exploiting properly. Combined with the other metrics in our reward function, this component does reward the best performing CNNs, making our agent better at differentiating between the top-performing neural networks.



(a) Scatter plot of generated networks

(b) Q-Learning performance

Figure 18: An overview of both the number of trainable parameters, reward, ε, and Q-Learning performance when $R = GE'_{50}$.



(a) Scatter plot of generated networks

(b) Q-Learning performance

Figure 19: An overview of both the number of trainable parameters, reward, ε, and Q-Learning performance when $R = t'$.