Reinforcement Learning-based Design of Side-channel Countermeasures

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

Delft University of Technology, The Netherlands

Abstract. Deep learning-based side-channel attacks are capable of breaking targets protected with countermeasures. What is more, the constant progress in the last few years is making the attacks ever more powerful where we require fewer traces to break a target. Unfortunately, to protect against such attacks, we still rely solely on methods developed to protect against generic attacks. The works that consider the protection perspective are few and usually based on the adversarial examples concepts, which are not always easy to translate to real-world hardware implementation.
In this work, we ask whether we can develop combinations of countermeasures that protect against side-channel attacks. We consider several well-known hiding countermeasures and use the reinforcement learning paradigm to design specific sets of countermeasures that show resilience against deep learning-based attacks. Our results show it is possible to enhance the target resilience significantly, up to a point where deep learning-based attacks cannot obtain the secret information. At the same time, we consider the cost of implementing such countermeasures to balance security and implementation costs.

Keywords: Reinforcement learning, Side-channel analysis, Countermeasures, Deep learning, Hiding

1 Introduction

Deep learning is a very powerful option for profiling side-channel analysis (SCA). In profiling SCA, we assume an adversary with access to a clone device under attack. Using that clone device, the attacker builds a model that is used to attack the target. This scenario maps perfectly to supervised machine learning, where first, a model is trained (profiling phase) and then tested on previously unseen examples (attack phase). While other than deep learning techniques also work well in profiling SCA (e.g., random forest or support vector machines), deep learning (deep neural networks) is commonly considered as the most powerful direction. This is because deep neural networks 1) do not require pre-processing, which means we can use raw traces, and 2) can break protected implementations, which seems to be much more difficult with simpler machine learning techniques or template attack. As such, the last few years brought several research works that report excellent attack performance and breaking of targets in a (commonly) few hundred traces. What is more, attack improvements are regularly appearing as many new results from the machine learning domain can be straightforwardly applied to improve the side-channel attacks, see, e.g., [17,22,16,14]. Simultaneously, there are only sporadic improvements from the defense perspective, and almost no research aimed to protect against deep learning-based SCA.

We consider this an important research direction. Indeed, if deep learning attacks are the most powerful ones, then an intuitive direction should be to design countermeasures against such attacks. Unfortunately, this is also a much more difficult research perspective. We can find several reasons for it:

– As other domains do not consider countermeasures in the same shape as in SCA, it is not straightforward to use the knowledge from other domains.
– While adversarial machine learning is an active research direction and intuitively, adversarial examples are a good defense against deep learning-based SCA, it is far from trivial to envision how such defenses would be implemented in cryptographic hardware. Additionally, adversarial examples commonly work in the amplitude domain but not in the time domain.
– It can be easier to attack than to defend [1]. Confirming that an attack is successful is straightforward as it requires assessing how many traces are needed to break the implementation. Confirming that a countermeasure works would, in an ideal case, require testing against all possible attacks (which is not possible).

There are only a few works considering countermeasures against machine learning-based SCA to the best of our knowledge. Inci et al. used adversarial learning as a defensive tool to obfuscate and mask side-channel information (concerning micro-architectural attacks) [8]. Picek et al. considered adversarial examples as a defense against power and EM side-channel attacks [15]. While they reported the defense works, they leave it open how would such a countermeasure be implemented. Finally, Gu et al. used an adversarial-based countermeasure that inserts noise instructions into code [6]. Interestingly, the authors report their approach also works against classical side-channel attacks.

In this paper, while we aim to reach the same goal as those works, we consider a radically different direction. We do not generate adversarial examples (although one could call our countermeasure to be adversarial), but we optimize the combinations of well-known SCA hiding countermeasures. More precisely, we use the reinforcement learning paradigm to find a combination of hiding countermeasures that makes deep learning-based SCA difficult to succeed. We emphasize that we simulate the countermeasures to assess their influence on a dataset [2]. Our combinations of countermeasures work in both amplitude and time domain and could be implemented in real-world targets. We conduct experiments considering three datasets and two leakage models, where our results

[1] In the context of masking and hiding countermeasures.

[2] This is why we concentrate on hiding countermeasures as it is easier to simulate hiding than masking. Still, as we attack datasets protected with masking, we consider both countermeasure categories covered.

indicate the random delay interrupt countermeasure as the key ingredient of strong resilience against deep learning-based SCA. Our main contributions are:

1. We propose a novel reinforcement learning approach to construct combinations of hiding countermeasures making deep learning-based SCA difficult to succeed.
2. We motivate and develop custom reward functions for countermeasure selection to increase the SCA resilience.
3. We conduct extensive experimental analysis considering four countermeasures, three datasets, and two leakage models.
4. We report on a number of countermeasures that perform the best and worst for the selected profiling model and datasets.

We plan to release our source code upon the acceptance of the paper (source code is available through program chairs if required).

2 Preliminaries

2.1 Notation

– Calligraphic letters ($\mathcal{X}$) denote sets and the corresponding upper-case letters (X) 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.

– A dataset $\mathbf{T}$ is a collection of traces (measurements). Each trace $\mathbf{t}_i$ is associated with an input value (plaintext or ciphertext) $\mathbf{d}_i$ and a key candidate $\mathbf{k}_i$. Here, $k \in \mathcal{K}$ and k^* represents the correct key. As common in profiling SCA, we divide the dataset into three parts: a profiling set of N traces, a validation set of V traces, and an attack set of Q traces.

– We denote the vector of learnable parameters in our profiling models as $\boldsymbol{\theta}$ and the set of hyperparameters defining the profiling model f as $\mathcal{H}$.

2.2 Deep Learning and Profiling Side-channel Analysis

We consider the supervised learning task where the goal 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. There is a natural mapping between supervised learning and profiling SCA. Supervised learning has two phases: training and test. The training phase corresponds to the SCA profiling phase, and the testing phase corresponds to the SCA attack phase. The profiling SCA runs under the following setup:

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

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

– 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.

Probabilistic deep learning algorithms output a matrix that denotes the probability that a certain measurement should be classified into a specific class. Thus, the result is a matrix P with dimensions equal to $Q \times c$. The probability $S(k)$ for any key byte candidate k is the maximum log-likelihood distinguisher:

$$S(k) = \sum_{i=1}^{Q} \log(\mathbf{p}_{i,v}). \tag{2}$$

The value $\mathbf{p}_{i,v}$ represents the probability that a specific class v is predicted. The class v is obtained from the key and input through a cryptographic function and a leakage model l.

From the matrix P, it is straightforward to obtain the accuracy of the model f. Still, in SCA, an adversary is not interested in predicting the classes in the attack phase but obtaining the secret key k^*. Thus, to estimate the difficulty of breaking the target, it is common to use metrics like guessing entropy (GE) [18]. There, 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 represents the average position of k^* in $\mathbf{g}$.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are commonly used neural networks in many domains, including SCA. They commonly consist of three types of layers: convolutional layers, pooling layers, and dense layers, also known as fully connected layers. The convolution layer 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. It is common to consider convolution and pooling layers to form a convolution block. The dense layers compute either the hidden activations or the class scores. In this work, we consider CNN-based SCA only, as previous results indicate that CNNs are the most powerful option for deep learning-based SCA.

2.4 Side-channel Countermeasures

As already discussed, there are many side-channel attacks on cryptographic implementations. To make the attacks more difficult to succeed, it is common to protect the implementation with countermeasures. Countermeasures aim to destroy the statistical link between intermediate values and traces (e.g., power consumption or EM emanation). There are two main categories of countermeasures for SCA: masking and hiding.

In masking, a random mask is generated to conceal every intermediate value. More precisely, random masks are used to remove the correlation between the measurements and the secret data. In general, there are two types of masking: Boolean masking and arithmetic masking.

On the other hand, the goal of hiding is to make measurements looking random or constant. Hiding decreases the signal-to-noise ratio (SNR) only. Hiding can happen in the amplitude (e.g., adding noise) and time (e.g., desynchronization, random delay interrupts, jitter) dimensions. Our work considers hiding countermeasures only as masking is difficult to simulate, and there are not many options for reinforcement learning to tune if considering masking.

2.5 Datasets and Leakage Models

ASCAD Datasets. The first two datasets are versions of the ASCAD database [1]. Both datasets contain 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 three). The datasets are available at `https://github.com/ANSSI-FR/ASCAD`.

The first dataset version has a fixed key (thus, the key is the same in profiling and attack set). This dataset consists of 50 000 traces for profiling and 10 000 for the attack. From 50 000 traces in the profiling set, we use 45 000 traces for profiling and 5 000 for validation. Each trace has 700 features (preselected window).

The second version has random keys, with 200 000 traces for profiling and 100 000 for the attack. We use 5 000 traces from the attack set for validation (note that the attack set has a fixed but a different key from the profiling set). Each trace has 1 400 features (preselected window).

CHES CTF Dataset. This dataset consists of masked AES-128 encryption running on a 32-bit STM microcontroller, and we attack the first key byte. This dataset is available at `https://chesctf.riscure.com/2018/news`. We use 45 000 traces for the training set (the training set has a fixed key). The attack set has 5 000 traces and uses a different key from the training set. We use 2 500 traces from the attack set for validation. CHES CTF trace sets contain the power consumption of the full AES-128 encryption, with a total number of 650 000 features per trace. The raw traces were pre-processed in the following way. First, a window resampling is performed, and later we concatenated the trace intervals representing the processing of the masks (beginning of the trace) with the first samples (processing of S-boxes) located after an interval without any particular activity (flat power consumption profile). Each trace has 2 200 features.

Leakage Models.
- The Hamming weight (HW) leakage model - the attacker assumes the leakage proportional to the sensitive variable's Hamming weight. As we consider the AES cipher with 8-bit S-boxes, this leakage model results in nine classes for a single key byte (values from 0 to 8).

– The Identity (ID) leakage model - the attacker considers the leakage in the form of an intermediate value of the cipher. As we consider the AES cipher with 8-bit S-boxes, this leakage model results in 256 classes for a single key byte (values from 0 to 255).

2.6 Reinforcement Learning

Reinforcement learning (RL) aims to teach an agent how to perform a task by letting the agent experiment and experience the environment. There are two main categories of reinforcement learning algorithms: policy-based algorithms and value-based algorithms. Policy-based algorithms directly try to find this optimal policy. Value-based algorithms, however, try to approximate or find the value function that assigns state-action pairs a reward value. Note that 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 or simulated annealing can all be used for reinforcement learning without ever estimating value functions [19]. In this research, we only focus on Q-Learning, belonging to the value estimation category.

Compared with supervised and unsupervised machine learning, which are commonly adopted by the SCA community, reinforcement learning has fundamental differences. Supervised machine learning learns from a set of examples (input-output pairs) labeled with the correct answers. A benefit 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. In terms of unsupervised machine learning, the algorithm attempts to find some (hidden) structure within a dataset, while reinforcement learning aims to teach an agent how to perform a task through rewards and experiments [19].

Q-Learning Q-Learning was introduced in 1989 by Chris Watkins [20] with an aim not only to learn from the outcome of a set of state-action transitions but from each of them individually. Q-learning is a value-based algorithm that tries to estimate $q_*(s,a)$, the reward of taking action a in the state s under the optimal policy, by iteratively updating its stored q-value estimations using Eq. (3). The simplest 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. A depiction of the Q-learning algorithm is presented in Figure 1.

Eq. (3) is used to incorporate the obtained reward into the saved reward for the current state R_t. S_t and 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 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 to

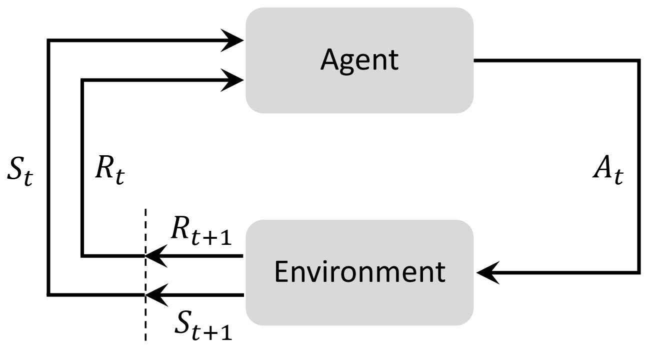


Fig. 1: The Q-learning concept. Here, an agent chooses an action A_t, based on the current state S_t, which affects the environment. This action is then given a reward R_{t+1} and leads to state S_{t+1}. After the reward for the current state is registered, the cycle starts again.

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}.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{3}$$

3 Related Works

We divide related works into two directions: improving deep learning-based SCA and improving the defenses against such attacks. In the first direction, from 2016 and the first paper using convolutional neural networks [10], there are continuous improvements in the attack performance. Commonly, such works investigate (note this is only a small selection of the papers):

– **the importance of hyperparameters and designing top-performing neural networks**. Benadjila et al. made an empirical evaluation of different CNN hyperparameters for the ASCAD dataset [1]. Perin and Picek explored the various optimizer choices for deep learning-based SCA [13]. Zaid et al. proposed a methodology to select hyperparameters related to the size of layers in CNNs [25]. To the best of our knowledge, this is the first methodology to build CNNs for SCA. Wouters et al. [21] improved upon the work from Zaid et al. [25] and showed it is possible to reach similar attack performance with significantly smaller neural network architectures. Wu et al. used Bayesian optimization to find optimal hyperparameters for multilayer perceptron and convolutional neural network architectures [22]. Rijsdijk et al. used reinforcement learning to design CNNs that exhibit strong attack performance, and additionally that have a small number of trainable parameters [17]. Our reinforcement learning setup is inspired by the one presented

here, especially the reward function part [3] In efforts to improve the attack performance, some authors also proposed custom elements for neural networks for SCA. For instance, Zaid et al. [24], and Zhang et al. [26] introduced new loss functions that improve the attack performance.
- **well-known techniques from the machine learning domain to improve the performance of deep learning-based attacks**. Cagli et al. showed how CNNs could defeat jitter countermeasure, and they used data augmentation to improve the attack process [2]. Kim et al. constructed VGG-like architecture that performs well over several datasets, and they use regularization in the form of noise addition to the input [9]. Perin et al. showed how ensembles could improve the attack performance even when single models are only moderately successful [12]. Wu et al. used the denoising autoencoder to remove the countermeasures from measurements to improve the attack performance [23]. Perin et al. considered the pruning technique and the lottery ticket hypothesis to make small neural networks that reach top attack performance [14].
- **explainability and interpretability of results**. Hettwer et al. investigated how to select points of interest for deep learning by using three deep neural network attribution methods [7]. Masure et al. used gradient visualization to discover where the sensitive information leaks [11].

On the other hand, the domain of countermeasures' design against machine learning-based SCA is much less explored [4]. Indeed, to the best of our knowledge, there are only three works considering this perspective as briefly discussed in Section 1. Interestingly, all those works investigated adversarial examples [8,15,6] and do not consider more "standard" SCA countermeasures. Additionally, it is not clear how all countermeasures proposed there would be implemented in real-world settings.

4 The Reinforcement Learning-based Countermeasure Selection Framework

This section provides details on our experimental setup, reward functions, and the deep learning models we consider.

4.1 General Setup

We propose a Tabular Q-Learning algorithm based on MetaQNN that can select countermeasures, including their parameters, to simulate their effectiveness on an existing dataset against an arbitrary neural network. To evaluate the effectiveness of the countermeasures, we use guessing entropy. There are several aspects to consider if using MetaQNN:

[3] The authors mention they conducted a large number of experiments to find a reward function that works well for different datasets and leakage models, so we decided to use the same reward function.

[4] Many works consider the development of SCA countermeasures, but not specifically against deep learning approaches.

1. We need to develop an appropriate reward function that considers particularities of the SCA domain. Thus, considering only machine learning metrics would not suffice.
2. MetaQNN uses a fixed α (learning rate) for Q-Learning while using a learning rate schedule where α decreases either linearly or polynomially are the normal practice [5].
3. One of the shortcomings of MetaQNN is that it requires significant computational power and time to explore the search space properly. As we consider several different countermeasures where each has its hyperparameters, this results in a very large search space.

Selecting the right countermeasures and their parameters is modeled as a Markov Decision Process (MDP). Specifically, each state has a transition towards an accepting state with the currently selected countermeasures. Each countermeasure can only be applied once per Q-Learning iteration, so the resulting set of chosen countermeasures can be empty (no countermeasure being added) or contain up to four different countermeasures in any order.[5] One may consider that with the larger number of countermeasures being added to the traces, the more difficult the secret information to be retrieved by the side-channel attacks. Although this statement is true, the implementation of the countermeasure is not without any cost. Indeed, some software-based countermeasures add overhead in the execution efficiency (i.e., dummy executions), while others add overhead in total power consumption (i.e., dedicated noise engine). To select optimal countermeasure combinations with a limited burden on the device, a cost function that can approximate the implementation costs would be useful to balance the strength of the countermeasure implementation and the security of the device, which is also a perfect candidate as a reward function to guide the Q-learning process.[6] Following this, we design a cost function associated with each of the countermeasures, where the value also depends on the chosen countermeasure's configuration. The total cost of the countermeasure set, c_{total}, is defined in Eq. (4).

$$c_{total} = \sum_{i=1}^{|C|} c_i. \tag{4}$$

Here, C represents the set of applied countermeasures, and c_i is the cost of the individual countermeasure defined differently for each countermeasure. Based on the values chosen by Wu et al. [23] for the ASCAD fixed key dataset, we set the total cost budget c_{max} to five. Indeed, c_{max} set the upper limit of the applied countermeasure so that the selected countermeasure is in a reasonable range. Only countermeasure configurations within the remaining budget are selectable

[5] The countermeasures set is an ordered set based on the order that the RL agent selected them. Since the countermeasures are applied in this order, sets with the same countermeasures but a different ordering are treated as disjoint.

[6] While we try to base the costs on real-world implications of adding each of the countermeasures in a chosen configuration, translating the total cost back to a real-world metric is nontrivial.

by the Q-Learning agent. In case the countermeasures are successful in defeating the attack (GE does not reach 0 within the configured number of attack traces), any leftover budget is used as a component of the reward function. By evaluating the reward function, we can find the best-performing countermeasure combinations, together with their settings, to protect the device from the SCA with the lowest budget.

In terms of types of countermeasures we evaluated, four types of countermeasures, desynchronization, uniform noise, clock jitter, and random delay interrupt (RDI), are analyzed and applied to the original dataset. The countermeasures are all applied a-posteriori to the chosen dataset in our experiments. Note that the implementations of the countermeasure are based on the countermeasure designs from Wu *et al.* [23].[7] The detailed implementation and design of each countermeasure's cost function are discussed in the following sections. As a reference, the pseudocodes for each implemented countermeasure are available in Section C.

Desynchronization Well-synchronized traces can significantly improve the correlation of the intermediate data and trace values, thus enhancing the efficiency of retrieving the secret information. Naturally, adding desynchronization to the traces is applied as a countermeasure to defeat the side-channel attack. Different from the clock jitter or RDIs that introduces the misalignment locally, desynchronization adds misalignment globally. For instance, we can break the alignment of two continuous encryptions by adding random executions between them. Aligned with the real-world implementations, we considered desynchronization as a countermeasure and add it as a-posteriori to the traces. For each trace in the dataset, we draw a number uniformly between 0 and the chosen maximum desynchronization and shift the trace by that number of features. In terms of the cost for desynchronization, Wu *et al.* showed that a maximum desynchronization of 50 already greatly improves the attack's difficulty. Even the CNN used there needs ten times the number of traces to retrieve the secret key. This leads us to set the desynchronization level ($desync\_level$) ranges from 5 to 50 in a step of 5. The cost calculation for desynchronization is defined in Eq. (5). Note that the maximum c_{desync} is five, which matches the c_{max} we defined as the total cost of countermeasures (which is why c_{desync} needs to be divided by ten).

$$c_{desync} = \frac{desync\_level}{10}. \tag{5}$$

Uniform noise Several sources, such as the transistor, data buses, the transmission line to the record devices such as oscilloscopes, or even the work environment, introduce noise to the amplitude domain. Adding uniform noise amounts

[7] Some of these countermeasures generate traces of varying length. To make them all of the same length, the traces shorter than the original are padded with zeroes, while any longer traces are truncated back to the original length.

to adding a uniformly distributed random value to each feature. To make sure the addition of the noise causes a similar effect on different datasets, we set the maximum *noise_level* based on the dataset variation defined by Eq. 6.

$$max\_noise\_level = \frac{\sqrt{Var(T)}}{2}. \tag{6}$$

Here, T denotes the measured leakage traces. Then, *max_noise_level* is multiplied with a *noise_factor* parameter, ranging from 0.1 to 1.0 with steps of 0.1, to control the actual noise level introduced to the traces. Since the *noise_factor* is the only adjustable parameter, we define the cost of the uniform noise in Eq. (7) to make sure that the the maximum c_{noise} equals to c_{max}.

$$c_{noise} = noise\_factor \times 5. \tag{7}$$

Clock Jitter One way of implementing clock jitters is by introducing the instability in the clock [3]. While desynchronization introduces randomness globally in the time domain, the introduction of clock jitters increases each sampling point's randomness, thus increasing the alignment difficulties. When applying the clock jitter countermeasure to the ASCAD dataset, Wu *et al.* chose eight as the jitter level, but none of the attacks managed to retrieve the key in 10 000 traces. However, we could observe that CNN and template attack do reduce the average key rank to below 50 at that point. Therefore, we decide to tune the jitter level (*jitter_level*) with a maximum of eight. The corresponding cost function is defined in Eq. 8. In the following experiments, we set the *jitter_level* ranging from 2 to 8 in a step of 2. Again, maximum c_{jitter} value matches the c_{max} value we defined before.

$$c_{jitter} = jitter\_level \times 1.6. \tag{8}$$

Random Delay Interrupts (RDIs) Similar to clock jitter, RDIs introduce local desynchronization in the traces. We implement RDIs based on the floating mean method [4]. More specifically, we add RDI for each feature in each trace with a configurable probability. Moreover, if an RDI occurs for a trace feature, we select the delay length based on the A and B parameters, where A is the maximum length of the delay and B is a number $\leqslant A$. Since RDIs in practice are implemented using instructions such as *nop*, we do not simply flatten the simulated power consumption but introduce peaks with a configurable amplitude.

Since the RDI countermeasure has many adjustable parameters, it will, by far, have the most MDP paths dedicated to it, meaning that during random exploration, it is far more likely to select it as a countermeasure. To offset this, we reduce the number of configurable parameters by fixing the amplitude for RDIs based on the *max_noise_level* defined in Eq. 6 for each dataset. Furthermore, we add 1 to the cost of any random delay interrupt countermeasure, as shown in Eq. 9, defining the cost function for RDIs.

$$c_{rdi} = 1 + \frac{3 \times probability \times (A + B)}{2}, \tag{9}$$

where A ranges from 1 to 10, B ranges from 0 to 9, and *probability* ranges from 0.1 to 1 in a step of 1. We emphasize that we made sure the selected B value is never larger than A.

When looking at the parameters Wu *et al.* [23] used for random delay interrupts applied on the ASCAD fixed key dataset, $A = 5$, $B = 3$ and $probability = 0.5$, none of the chosen attack methods show any signs of converging on the correct key guess, even after 10 000 traces. With our chosen c_{rid}, this configuration cost equals seven, which we consider appropriate.

4.2 Reward Functions

To allow MetaQNN to be used for the countermeasure selection, we use a relatively complex reward function. This reward function incorporates the guessing entropy and is composed of four metrics: 1) t': the percentage of traces required to get the GE to 0 out of the fixed maximum attack 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) c': the percentage of countermeasures budget left over out of the fixed maximum budget parameter. The formal definitions of the first three metrics are expressed in Eqs. (10), (11), (12), and (13). We note this is the same reward function as used in [17].

$$t' = \frac{t_{max} - min(t_{max},\ \overline{Q}_{t_{GE}})}{t_{max}}. \tag{10}$$

$$GE'_{10} = \frac{128 - min(GE_{10}, 128)}{128}. \tag{11}$$

$$GE'_{50} = \frac{128 - min(GE_{50}, 128)}{128}. \tag{12}$$

$$c' = \frac{c_{max} - c_{total}}{c_{max}}. \tag{13}$$

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 attack traces. Since we reward countermeasure sets that manage to reduce the SCA performance, we incorporate the inverse of these metrics into our reward functions, as these metrics are appropriate in a similar setting [17]. Combining these three metrics allows us to assess the countermeasure set performance, even if the neural network model does not retrieve the secret key within the maximum number of attack traces. We incorporate these metrics inversely into our reward function by subtracting their value from their maximum value. Combined, these maximum values from which we subtract sum (multiplied by their weight in the reward function) to 2.5, as shown in Eq. (14).

In terms of the fourth metric c', recall C is the set of countermeasures chosen by the agent, and c_{total} equals five. We only apply this reward when the key retrieval is unsuccessful in t_{max} traces, as we do not want to reward small countermeasure sets for their size if they do not adequately decrease the SCA performance. Combining these four metrics, we define the reward function as in Eq. (14), which then gives us a total reward between 0 and 1. To better reward the countermeasure set performance, making the SCA neural networks require more traces for a successful break, a smaller weight is set on GE'_{50}.

$$R = \frac{1}{3} \times \begin{cases} 2.5 - t' - GE'_{10} - 0.5 \times GE'_{50}, & \text{if } t_{GE=0} < t_{max} \\ 2.5 - GE'_{10} - 0.5 \times GE'_{50} + 0.5 \times c', & \text{otherwise} \end{cases} \tag{14}$$

Note that we multiply the entire set of metrics by $\frac{1}{3}$ to normalize our reward function between 0 and 1. Furthermore, as already mentioned, we subtract the three SCA performance metrics from their combined maximum value of 2.5 to reward the countermeasure set performance. Finally, we only add the c' component of the reward function when the SCA cannot retrieve the secret key within the configured maximum number of traces t_{max} to avoid rewarding low-cost countermeasure sets that do not significantly decrease the SCA performance. While this reward function does look complicated, it is derived based on the results from [17] and our experimental tuning lasting several weeks. Still, we do not claim the presented reward function is optimal, but we claim it gives good results. Further improvements, especially from the perspective of the budget or the cost of a specific countermeasure, are always possible.

5 Experimental Results

To assess the performance of the selected set of countermeasures for each dataset and leakage model, we perform experiments with different CNN models. Those models are tuned for each dataset and leakage model combination but without considering hiding countermeasures that we simulate. Specifically, we use reinforcement learning to select the model's hyperparameter [17]. For every dataset and leakage model combination, we execute the search algorithm and select the top-performing models among over 2 500 iterations.

The details about the specific architectures can be found in Table 1. Note that Rijsdijk *et al.* [17] implemented two reward functions: one that only considers the attack performance, and the other that also considers the network size (small reward function). Aligned with that paper, we consider both reward functions, leading to two models being used for testing; the one denoted with RS is the models optimized with the small reward function. For all models, we use $he\_uniform$ and $selu$ as kernel initializer and activation function. We show the top-performing countermeasures for each dataset and profiling models in the paper; the worst-performing countermeasures are available in Section B in Appendix B. For all experiments, we denote countermeasure with the CM abbreviation.

Test models	Convolution (filter_number, size)	Pooling (size, stride)	Fully-connected layer
$ASCAD_{HW}$	Conv(16,100)	avg(25,25)	15+4+4
$ASCAD_{HW\_RS}$	Conv(2,25)	avg(4,4)	15+10+4
$ASCAD_{ID}$	Conv(128,25)	avg(25,25)	20+15
$ASCAD_{ID\_RS}$	Conv(2+2+8, 75+3+2)	avg(25+4+2, 25+4+2)	10+4+2
$ASCAD\_R_{HW}$	Conv(4, 50)	avg(25, 25)	30+30+30
$ASCAD\_R_{HW\_RS}$	Conv(8, 3)	avg(25, 25)	30+30+20
$ASCAD\_R_{ID}$	Conv(128, 3)	avg(75, 75)	30+2
$ASCAD\_R_{ID\_RS}$	Conv(4, 1)	avg(100, 75)	30+10+2
$CHES\ CTF_{HW}$	Conv(4, 100)	avg(4, 4)	15+10+10
$CHES\ CTF_{HW\_RS}$	Conv(2, 2)	avg(7, 7)	10

Table 1: CNN architectures used in the experiments [17].

5.1 ASCAD Fixed Key Dataset

Figure 2 shows the scatter plot results for the HW and ID leakage models for both the regular and RS CNN. The vertical red line indicates the highest Q-learning reward for the countermeasure set, which could not prevent the CNN from retrieving the key within the configured 2 000 attack traces. Notably, a sharp line can be found on the right side of the Q-Learning reward plots, which is solely due to the c' component of the reward function. Although the selected CNNs can retrieve the secret key when no countermeasures were applied ($c' = 0$) for all experiments with both HW and ID leakage models, as soon as any countermeasure is applied, the attack becomes unsuccessful with 2 000 attack traces. Indeed, we observe that only very few of the countermeasures seem to be inefficient in defeating the deep learning attacks from the result plots.

For the experiments shown in Figure 2a, the top countermeasures for ASCAD using different profiling models are listed in Table 2. Notably, the best countermeasure set in terms of performance and cost for this CNN consists of desynchronization with a level equal to ten. Interestingly, a desynchronization of 10 combined with uniform noise with noise factor 0.6, presented in Figure 5, Section B, appears as one of the worst-performing countermeasure sets with $GE_{10} = 82.19$ and $GE_{50} = 45.62$. This further highlights that manual selection of countermeasures might lead to unexpected results and that uniform noise can help in increasing SCA attack resilience, especially in the presence of countermeasures [9]. The rest of the top 20 countermeasure sets include or solely consist of random delay interrupts. This observation is also applied to other profiling models and ID leakage models. The amplitude for RDI is fixed for each dataset, as explained in Section 4.1. In terms of the parameters of RDIs, B stays zero for all three profiling models, indicating that the length of RDIs is solely determined by A. Indeed, B varies the mean of the number of added RDIs and enhances the difficulties in learning from the data. However, a larger B value would also increase the countermeasure cost, which is against the reward function's principle. From Table 2, we can observe both low values of A and *probability* being applied to the RDIs countermeasure, indicating the success of our framework in finding countermeasure with high performance and low cost.

Model	Reward	Countermeasures	c'
$ASCAD_{HW}$	0.967	Desync(desync_level=10)	1.00
$ASCAD_{HW\_RS}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=12.88)	1.15
$ASCAD_{ID}$	0.957	RDI(A=2,B=0,probability=0.10,amplitude=12.88)	1.30
$ASCAD_{ID\_RS}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=12.88)	1.15

Table 2: Best performing countermeasures for the ASCAD with fixed key dataset.

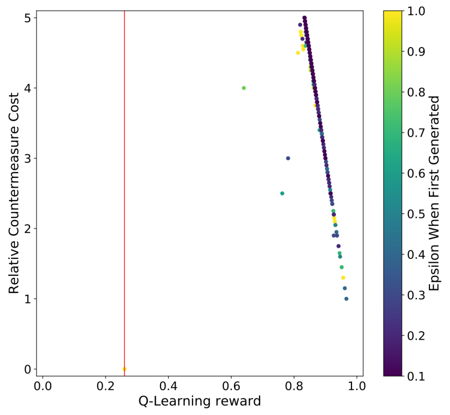
Next, we compare the general performance of the countermeasure sets between CNNs designed for the HW and ID leakage model. We observe that the ID model appears to be at least a little better at handling countermeasures. Specifically, for the ID leakage model CNNs, the variance of the countermeasures' Q-Learning reward is higher, indicating that the ID model CNNs can better handle countermeasures, making the countermeasure selection more important. This observation is confirmed by the c' value listed in Table 2: to reach a similar level of the reward value, the countermeasures are implemented with a greater cost.

Considering the time required to run the reinforcement learning, we observe we require around 200 hours on average, which is double the time required by Rijsdijk et al. when finding neural networks that perform well [17].

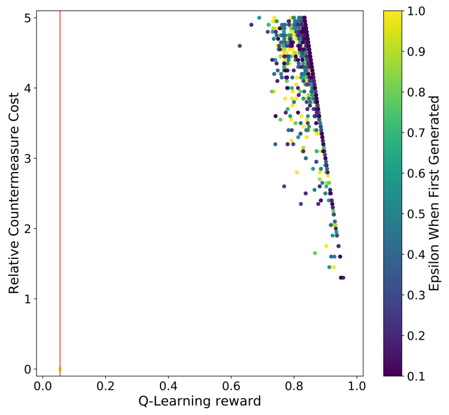
5.2 ASCAD Random Keys Dataset

The scatter plot results for both the HW and ID model for both the regular and RS CNN are listed in Figure 3. Aligned with the observation for the ASCAD fixed key dataset, the vertical red line in the plots is far away from the dots in the plot, indicating that the countermeasure's addition effectively increases side-channel attack difficulty. Furthermore, we again see the sharp line on the right side of the Q-Learning reward, which is caused by the c' component of the reward function.

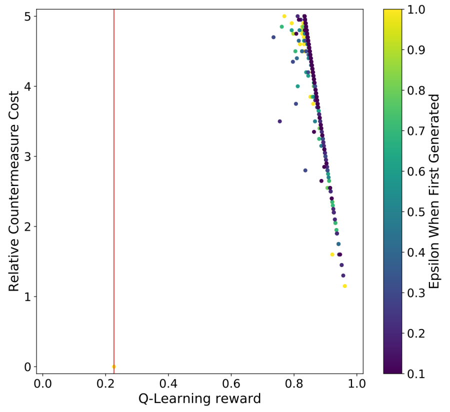
Compared with the ASCAD results for both leakage models (Figure 2), we see a greater variation of the individual countermeasure implementations: even with the same countermeasure cost, a different combination of countermeasures and their corresponding setting may lead to unpredictable reward values. Fortunately, with the RL-based countermeasure selection scheme, we see this tendency



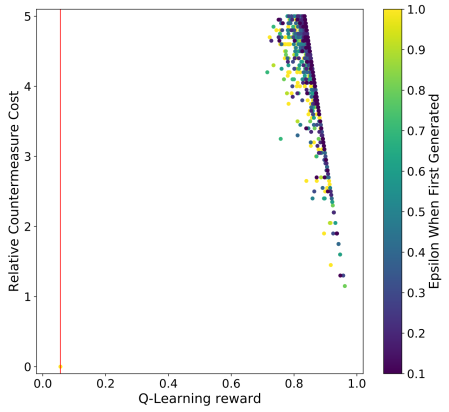
(a) CM ASCAD Fixed Key HW Model (Time consumption: 192 hours)



(b) CM ASCAD Fixed Key ID Model (Time consumption: 204 hours)

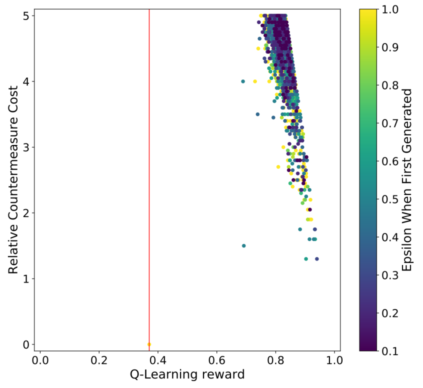


(c) CM ASCAD Fixed Key HW Model (RS) (Time consumption: 196 hours)

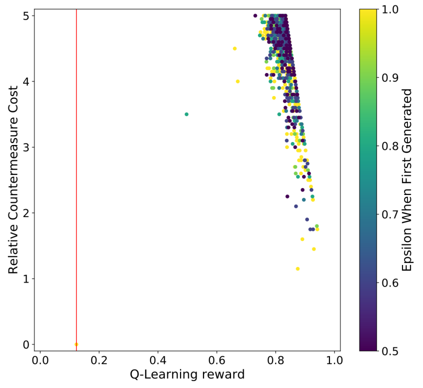


(d) CM ASCAD Fixed Key ID Model (RS) (Time consumption: 198 hours)

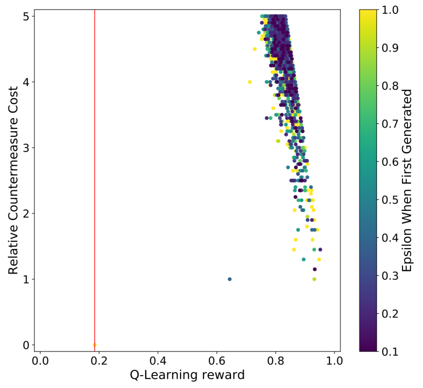
Fig. 2: An overview of the countermeasure cost, reward, and the ε value a countermeasure combination set was first generated for the ASCAD with fixed key dataset experiments. The red lines indicate the countermeasure set with the highest reward for which the GE reached 0 within 2 000 traces.



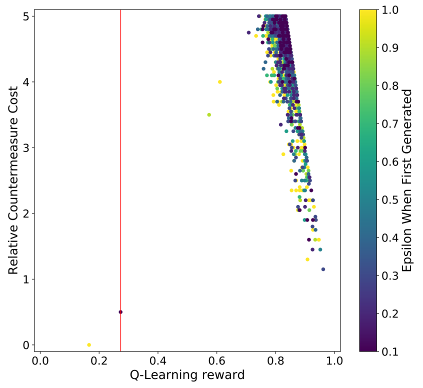
(a) CM ASCAD Random Keys HW Model
(Time consumption: 280 hours)



(b) CM ASCAD Random Keys ID Model
(Time consumption: 48 hours)



(c) CM ASCAD Random Keys HW Model (RS)
(Time consumption: 296 hours)



(d) CM ASCAD Random Keys ID Model (RS)
(Time consumption: 309 hours)

Fig. 3: An overview of the countermeasure cost, reward, and the ε value a countermeasure combination set was first generated for the ASCAD with random keys dataset experiments. The red lines indicates the countermeasure set with the highest reward for which the GE reached 0 within 2 000 traces.

and can better select the countermeasures' implementation with a limited budget. Finally, when comparing the HW and ID leakage models, we observe that the later leakage model is more effective in defeating the countermeasure. In other words, to protect the essential execution which leaks the ID information, more effort may be required to implementing countermeasures. The top-performing countermeasures for different profiling models are listed in Table 3. From the results, RDIs again become the most effective one among all of the considered countermeasures. The RDI amplitude is fixed at 16.95 for this dataset, as explained in Section 4.1. Interestingly, the countermeasures are implemented with higher cost when compared with the one used for ASCAD with a fixed key. The reason could be that training with random- ey traces enhances the generalization of the profiling model. What is more, we also observe that we require significantly longer time to run the reinforcement learning framework: on average, 300 hours, which is more than 12 days of computations. Interestingly, we see an outlier with the ASCAD random keys for the ID leakage model, where only 48 hours were needed for the experiments.

Model	Reward	Countermeasures	c'
$ASCAD\_R_{HW}$	0.940	RDI(A=1,B=0,probability=0.20,amplitude=16.95)	1.30
$ASCAD\_R_{HW\_RS}$	0.952	RDI(A=2,B=1,probability=0.10,amplitude=16.95)	1.45
$ASCAD\_R_{ID}$	0.942	RDI(A=5,B=0,probability=0.10,amplitude=16.95)	1.75
$ASCAD\_R_{ID\_RS}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=16.95)	1.15

Table 3: Best performing countermeasures for the ASCAD Random Keys dataset.

5.3 CHES CTF Dataset

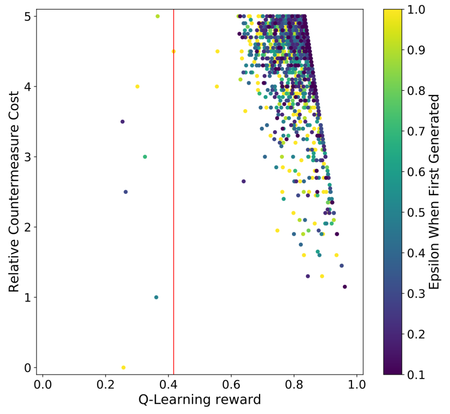
Finally, we test the CHES CTF dataset by adding different types of countermeasures. The results are presented in Figure 4. Note that CHES CTF leaks in HW only, and following this, we only attack the dataset with the HW leakage model. First, compared to the other two datasets, the highest Q-learning reward with GE equals zero with 2 000 traces (red line) becomes significantly higher (0.4), indicating a stronger CHES CTF vulnerability dataset towards deep learning attacks. This observation can also be confirmed when looking at the dots' distribution (representing different combinations of countermeasures) within the plot: for both tested models, compared with the other two datasets, we see a greater variation of the Q-learning reward with the same countermeasure costs. Nevertheless, with our RL-based countermeasure selection framework, the best countermeasure combination with the least cost can be found in the right corner of the graph.

Furthermore, we list the best countermeasure selected by the RL framework in Table 4. Aligned with the previous two datasets, RDIs become the most effective countermeasure for both profiling models. The RDI amplitude is fixed at 0.50 for this dataset, as explained in Section 4.1. In terms of countermeasure configurations, both parameters are kept in low values.

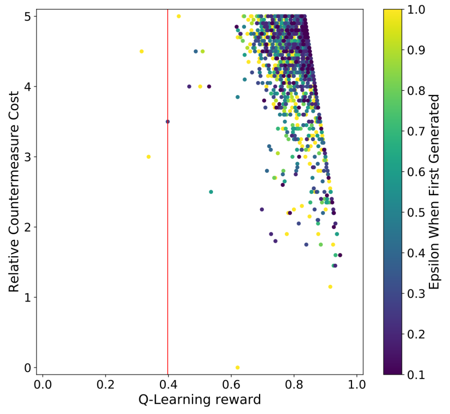
Interestingly, for all datasets and leakage models, we obtain RDI as the member of the countermeasure set performing the best. This indicates that RDI is very powerful, but it requires careful tuning of parameters. Indeed, Wu and Picek reported that clock jitter represents the biggest obstacle in the deep learning-based SCA [23], which indicates that the selection of RDI parameters was made in a suboptimal way.

Model	Reward	Countermeasures	c'
$CHES\ HW$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=0.50)	1.15
$CHES\ CTF_{HW\_RS}$	0.947	RDI(A=2,B=0,probability=0.20,amplitude=0.50)	1.60

Table 4: Best performing countermeasures for the CHES CTF dataset.



(a) CM CHES CTF HW Model (Time consumption: 125 hours)



(b) CM CHES CTF HW Model (RS) (Time consumption: 107 hours)

Fig. 4: An overview of the countermeasure cost, reward, and the ε value a countermeasure combination set was first generated for the CHES CTF dataset experiments. The red lines indicate the countermeasure set with the highest reward for which the GE reached 0 within 2 000 traces.

6 Conclusions and Future Work

This paper presents a novel approach to designing side-channel countermeasures based on reinforcement learning. More precisely, we consider four well-known types of countermeasures (one in the amplitude domain and three in the time domain), and we aim to find the best combinations of countermeasures within a specific budget. We conduct experiments on three datasets considering the HW and ID leakage models and report a number of countermeasure combinations providing significantly improved resilience against deep learning-based SCA. Our experiments show that the best performing countermeasure combinations use the random delay interrupt countermeasure, making it a natural choice for real-world implementations.

The experiments performed currently take significantly longer than might be necessary, as we generate a fixed number of unique countermeasure sets, while the chance to generate a unique countermeasure set towards the end of the experiments is significantly smaller (due to the lower ε). For example, we generated over 70 000 countermeasure sets to generate 1 700 unique countermeasure sets in the CHES CTF HW leakage model experiment. For future work, we plan to explore how to detect this behavior. Additionally, we plan to consider multilayer perceptron architectures and sets of countermeasures that work well for different datasets and leakage models. Finally, this work considers "static" neural network models and defenses that adapt. It would be interesting to consider a bi-level system where both neural networks models that are used in attacks and countermeasure sets are designed.

References

1. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ASCAD database. J. Cryptographic Engineering **10**(2), 163–188 (2020). https://doi.org/10.1007/s13389-019-00220-8, `https://doi.org/10.1007/s13389-019-00220-8`
2. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2017. pp. 45–68. Springer International Publishing, Cham (2017)
3. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 45–68. Springer (2017)
4. Coron, J., Kizhvatov, I.: An Efficient Method for Random Delay Generation in Embedded Software. In: Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. pp. 156–170 (2009)
5. Even-Dar, E., Mansour, Y.: Learning rates for q-learning. J. Mach. Learn. Res. **5**, 1–25 (Dec 2004)
6. Gu, R., Wang, P., Zheng, M., Hu, H., Yu, N.: Adversarial attack based countermeasures against deep learning side-channel attacks (2020)

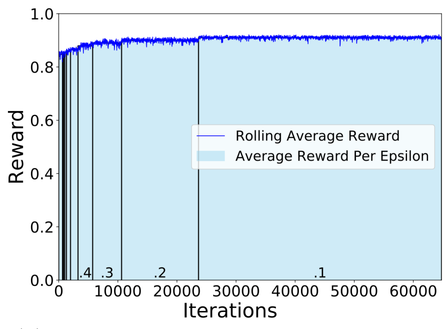
7. Hettwer, B., Gehrer, S., Güneysu, T.: Deep neural network attribution methods for leakage analysis and symmetric key recovery. In: Paterson, K.G., Stebila, D. (eds.) Selected Areas in Cryptography – SAC 2019. pp. 645–666. Springer International Publishing, Cham (2020)
8. Inci, M.S., Eisenbarth, T., Sunar, B.: Deepcloak: Adversarial crafting as a defensive measure to cloak processes. CoRR **abs/1808.01352** (2018), http://arxiv.org/abs/1808.01352
9. Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 148–179 (2019)
10. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking cryptographic implementations using deep learning techniques. In: International Conference on Security, Privacy, and Applied Cryptography Engineering. pp. 3–26. Springer (2016)
11. Masure, L., Dumas, C., Prouff, E.: Gradient visualization for general characterization in profiling attacks. In: Polian, I., Stöttinger, M. (eds.) Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11421, pp. 145–167. Springer (2019). https://doi.org/10.1007/978-3-030-16350-1_9, https://doi.org/10.1007/978-3-030-16350-1_9
12. Perin, G., Chmielewski, L., Picek, S.: Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 337–364 (Aug 2020). https://doi.org/10.13154/tches.v2020.i4.337-364, https://tches.iacr.org/index.php/TCHES/article/view/8686
13. Perin, G., Picek, S.: On the influence of optimizers in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/977 (2020), https://eprint.iacr.org/2020/977
14. Perin, G., Wu, L., Picek, S.: Gambling for success: The lottery ticket hypothesis in deep learning-based sca. Cryptology ePrint Archive, Report 2021/197 (2021), https://eprint.iacr.org/2021/197
15. Picek, S., Jap, D., Bhasin, S.: Poster: When adversary becomes the guardian – towards side-channel security with adversarial attacks. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2673–2675. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363284, https://doi.org/10.1145/3319535.3363284
16. Ramezanpour, K., Ampadu, P., Diehl, W.: Scarl: Side-channel analysis with reinforcement learning on the ascon authenticated cipher (2020)
17. Rijsdijk, J., Wu, L., Perin, G., Picek, S.: Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2021/071 (2021), https://eprint.iacr.org/2021/071
18. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Joux, A. (ed.) Advances in Cryptology - EUROCRYPT 2009. pp. 443–461. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
19. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, 2 edn. (2018), http://incompleteideas.net/book/the-book.html
20. Watkins, C.J.C.H.: Learning from delayed rewards. Phd thesis, University of Cambridge England (1989)

21. Wouters, L., Arribas, V., Gierlichs, B., Preneel, B.: Revisiting a methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 147–168 (Jun 2020). https://doi.org/10.13154/tches.v2020.i3.147-168, `https://tches.iacr.org/index.php/TCHES/article/view/8586`
22. Wu, L., Perin, G., Picek, S.: I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/1293 (2020), `https://eprint.iacr.org/2020/1293`
23. Wu, L., Picek, S.: Remove some noise: On pre-processing of side-channel measurements with autoencoders. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 389–415 (Aug 2020). https://doi.org/10.13154/tches.v2020.i4.389-415, `https://tches.iacr.org/index.php/TCHES/article/view/8688`
24. Zaid, G., Bossuet, L., Dassance, F., Habrard, A., Venelli, A.: Ranking loss: Maximizing the success rate in deep learning side-channel analysis. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(1), 25–55 (2021). https://doi.org/10.46586/tches.v2021.i1.25-55, `https://doi.org/10.46586/tches.v2021.i1.25-55`
25. Zaid, G., Bossuet, L., Habrard, A., Venelli, A.: Methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(1), 1–36 (Nov 2019). https://doi.org/10.13154/tches.v2020.i1.1-36, `https://tches.iacr.org/index.php/TCHES/article/view/8391`
26. Zhang, J., Zheng, M., Nan, J., Hu, H., Yu, N.: 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). https://doi.org/10.13154/tches.v2020.i3.73-96, `https://tches.iacr.org/index.php/TCHES/article/view/8583`

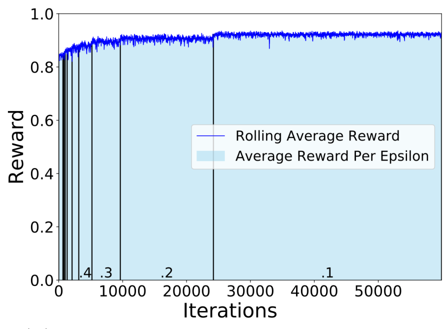
A Q-learning performance

In Figure 5c, we show the rolling average of the Q-learning reward and the average Q-learning reward per epsilon for the ASCAD fixed key dataset. As can be seen, the reward value for countermeasure gradually increases when more iteration is performed, indicating that the agent is learning from the environment and becoming more capable of finding effective countermeasure settings with a low cost. Then, the reward value is saturated when ε reaches 0.1, meaning that the agent is well trained and constantly finds well-performing countermeasures. One may notice that the number of iterations performed is significantly higher than the configured 1 700 iterations. This is because we only count an iteration when we generated a countermeasure set that was not generated before.

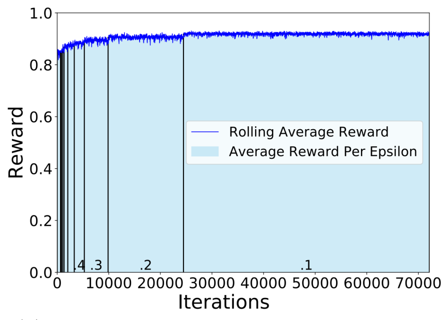
The rolling average of the Q-learning reward and the average Q-learning reward per ε for the ASCAD random keys dataset are given in Figure 6. Interestingly, at the beginning of Figure 6a, we see a significant drop in Q-learning reward, followed by a rapid increase in the ε update from 0.4 to 0.3. A possible explanation could be that the model we used is powerful in defeating the selected countermeasures at the early learning stage. Still, the algorithm managed to learn from each interaction, finally selecting powerful countermeasures.



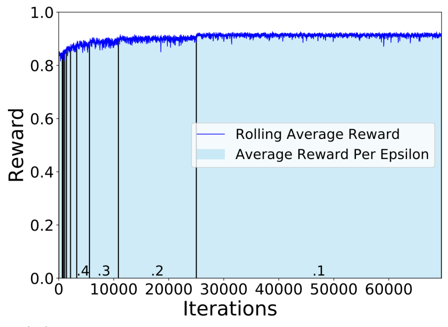
(a) CM ASCAD Fixed Key HW Model



(b) CM ASCAD Fixed Key ID Model



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

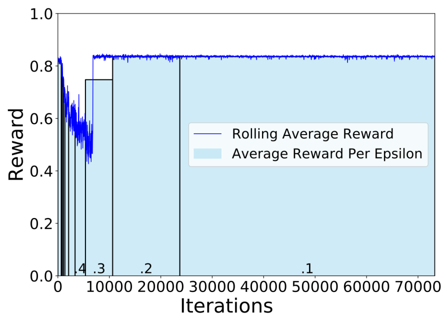


(d) CM ASCAD Fixed Key ID Model (RS)

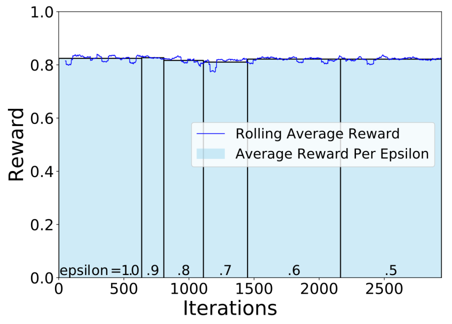
Fig. 5: An overview of the Q-Learning performance for the ASCAD with fixed key dataset experiments. The blue line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a countermeasure set. The bars in the graph indicate the average Q-Learning reward for all countermeasure sets generated during that ε.

In contrast, selecting countermeasure to defeat $ASCAD\_R_{ID}$ is an easy task: the reward value reaches above 0.8 at the very beginning, and it stops increasing regardless of the number of iterations. Since each test consumes 300 hours on average, we stopped the tests after around 3 000 iterations. For Figures 6c and 6d, we observe similar performance as in the ASCAD with the fixed key dataset: the RL algorithm is constantly learning. The highest reward value is obtained when ε reaches the minimum.

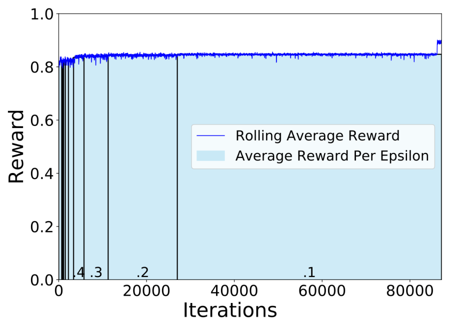
Finally, we present the rolling average of the Q-learning reward and the average Q-learning reward per ε for the CHES CTF dataset in Figure 7. Aligned with the previous two datasets, we see a gradual increase till ε reaches 0.1. Then, our algorithm constantly outputs well-performing countermeasures.



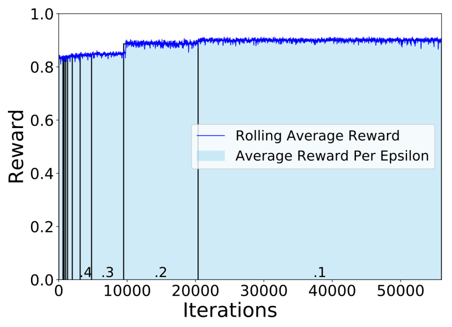
(a) CM ASCAD Random Keys HW Model



(b) CM ASCAD Random Keys ID Model



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



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

Fig. 6: 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 countermeasure set. The bars in the graph indicate the average Q-Learning reward for all countermeasure sets generated during that ε.

B Worst Performing Countermeasures

In this section, we show the worst performing sets of countermeasures for each dataset and leakage model. Additionally, we show both reward function versions: the regular one and with an added reward for small neural networks.

In Table 5, we give results for the ASCAD with the fixed key dataset, while in Table 6, we give results for the ASCAD with random keys dataset. Finally, in Table 7, we show the worst sets of countermeasures for the CHES CTF dataset and the HW leakage model.

C Pseudocode for Side-channel Countermeasures

Next, we give pseudocode for simulating each of the considered countermeasures. Algorithm 1 display the desynchronization countermeasure while Algorithm 2

Model	Reward	Countermeasures	c'
$ASCAD_{HW}$	0.640	Desync(desync_level=10) +UniformNoise(noise_factor=0.60,noise_scale=12.88)	4.00
$ASCAD_{HW\_RS}$	0.735	RDI(A=9,B=4,probability=0.10,amplitude=12.88) +ClockJitter(jitters_level=2) +UniformNoise(noise_factor=0.10,noise_scale=12.88)	4.70
$ASCAD_{ID}$	0.627	RDI(A=8,B=6,probability=0.10,amplitude=12.88) +Desync(desync_level=15)	4.60
$ASCAD_{ID\_RS}$	0.715	RDI(A=2,B=0,probability=0.40,amplitude=12.88) +UniformNoise(noise_factor=0.20,noise_scale=12.88) +Desync(desync_level=10)	4.20

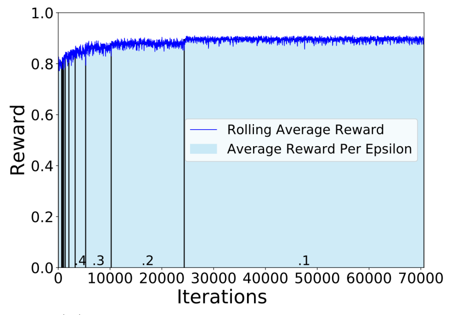
Table 5: Worst performing countermeasures for the ASCAD with fixed key dataset.

Model	Reward	Countermeasures	c'
$ASCAD\_R_{HW}$	0.689	UniformNoise(noise_factor=0.70,noise_scale=16.95) +Desync(desync_level=5)	4.00
$ASCAD\_R_{HW\_RS}$	0.643	UniformNoise(noise_factor=0.20,noise_scale=16.95)	1.00
$ASCAD\_R_{ID}$	0.497	UniformNoise(noise_factor=0.70,noise_scale=16.95)	3.50
$ASCAD\_R_{ID\_RS}$	0.273	UniformNoise(noise_factor=0.10,noise_scale=16.95)	0.50

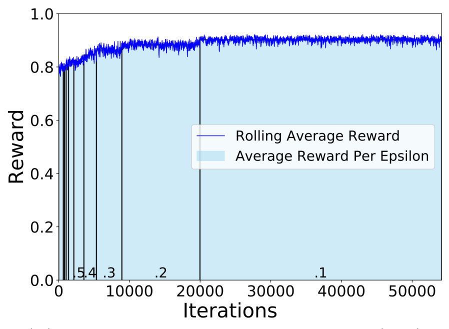
Table 6: Worst performing countermeasures for the ASCAD Random Keys dataset.

Model	Reward	Countermeasures	c'
$CHES\ CTF_{HW}$	0.254	UniformNoise(noise_factor=0.70,noise_scale=0.50)	3.50
$CHES\ CTF_{HW\_RS}$	0.315	RDI(A=2,B=0,probability=0.20,amplitude=0.50)	4.50

Table 7: Worst performing countermeasures for the CHES CTF dataset.



(a) CM CHES CTF HW Model



(b) CM CHES CTF HW Model (RS)

Fig. 7: 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 countermeasure set. The bars in the graph indicate the average Q-Learning reward for all countermeasure sets generated during that ε.

gives the code to simulate the uniform noise (we note that instead of uniform noise, others could be easily taken, e.g., Gaussian noise).

Algorithm 1 Add Desynchronization.

```
1: function ADD_DESYNC($trace, desync\_level$)
2:     $new\_trace \leftarrow []$                         ▷ container for new trace
3:     $level \leftarrow$randomNumber$(0, desync\_level)$
4:     $i \leftarrow 0$
5:     while $i + level < len(trace)$ do
6:         $new\_trace[i] \leftarrow traces[i + level]$     ▷ add desynchronization to the trace
7:         $i \leftarrow i + 1$
8:     return $new\_trace$
```

Algorithm 3 provides the pseudocode to simulate the addition of clock jitters, and finally, Algorithm 4 gives the pseudocode for the random delay interrupt countermeasure.

Algorithm 2 Add Uniform Noise.

```
1: function ADD_UNIFORM_NOISE($trace, range$)
2:     $new\_trace \leftarrow []$                                  ▷ container for new trace
3:     $i \leftarrow 0$
4:     while $i < len(trace)$ do
5:         $level \leftarrow$randomNumber$(-range, range)$
6:         $new\_trace[i] \leftarrow traces[i] + level$             ▷ add noise to the trace
7:         $i \leftarrow i + 1$
8:     return $new\_trace$
```

Algorithm 3 Add Clock Jitters.

```
1: function ADD_CLOCK_JITTERS($trace, clock\_jitters\_level$)
2:     $new\_trace \leftarrow []$                                  ▷ container for new trace
3:     $i \leftarrow 0$
4:     while $i < len(trace)$ do
5:         $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
6:         $r \leftarrow$randomNumber$(0, clock\_jitters\_level)$     ▷ level of clock jitters
7:         if $r < 0$ then
8:             $i \leftarrow i + r$                                ▷ skip points
9:         else
10:            $j \leftarrow 0$
11:            $average\_amplitude \leftarrow (trace[i] + trace[i + 1])/2$
12:            while $j < r$ do
13:                $new\_trace \leftarrow new\_trace.append(average\_amplitude)$    ▷ add points
14:                $j \leftarrow j + 1$
15:        $i \leftarrow i + 1$
16:    return $new\_trace$
```

Algorithm 4 Add Random Delay Interrupts.

```
1: function ADD_RDIS($traces, a, b, rdi\_amplitude$)
2:     $new\_trace \leftarrow []$                                  ▷ container for new trace
3:     $i \leftarrow 0$
4:     while $i < len(trace)$ do
5:         $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
6:         $rdi\_occurrence \leftarrow$randomNumber$(0, threshold * 2)$
7:         if $rdi\_occurrence > threshold$ then
8:             $m \leftarrow$randomNumber$(0, a - b)$
9:             $rdi\_num \leftarrow$randomNumber$(m, m + b)$     ▷ number of RDIs to be added
10:            $j \leftarrow 0$
11:            while $j < rdi\_num$ do                          ▷ add RDIs to the trace
12:                $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
13:                $new\_trace[i] \leftarrow new\_trace[i].append(trace[i] + rdi\_amplitude)$
14:                $new\_trace[i] \leftarrow new\_trace[i].append(trace[i + 1])$
15:                $j \leftarrow j + 1$
16:        $i \leftarrow i + 1$
17:    return $new\_trace$
```