NASCTY

Neuroevolution to Attack Side-Channel Leakages Yielding Convolutional Neural Networks

Schijlen, Fiske; Wu, Lichao; Mariot, Luca

DOI
10.3390/math11122616

Publication date
2023

Document Version
Final published version

Published in
Mathematics

mathematics

MDPI

Article

NASCTY: Neuroevolution to Attack Side-Channel Leakages Yielding Convolutional Neural Networks

Fiske Schijlen [1], Lichao Wu [1] and Luca Mariot [2,*]

[1] Cybersecurity Research Group, Delft University of Technology, Mekelweg 5, 2628 CD Delft, The Netherlands; fiske_schijlen@hotmail.com (F.S.); l.wu-4@tudelft.nl (L.W.)
[2] Semantics, Cybersecurity and Services Group, University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands
* Correspondence: l.mariot@utwente.nl

Abstract: Side-channel analysis (SCA) is a class of attacks on the physical implementation of a cipher, which enables the extraction of confidential key information by exploiting unintended leaks generated by a device. In recent years, researchers have observed that neural networks (NNs) can be utilized to perform highly effective SCA profiling, even against countermeasure-hardened targets. This study investigates a new approach to designing NNs for SCA, called *neuroevolution to attack side-channel traces yielding convolutional neural networks* (NASCTY-CNNs). This method is based on a genetic algorithm (GA) that evolves the architectural hyperparameters to automatically create CNNs for side-channel analysis. The findings of this research demonstrate that we can achieve performance results comparable to state-of-the-art methods when dealing with desynchronized leakages protected by masking techniques. This indicates that employing similar neuroevolutionary techniques could serve as a promising avenue for further exploration. Moreover, the similarities observed among the constructed neural networks shed light on how NASCTY effectively constructs architectures and addresses the implemented countermeasures.

Keywords: side-channel analysis (SCA); genetic algorithm (GA); neural network (NN); neural architecture search (NAS)

MSC: 94A60; 68P25; 68T07; 68W50

Citation: Schijlen, F.; Wu, L.; Mariot, L. NASCTY: Neuroevolution to Attack Side-Channel Leakages Yielding Convolutional Neural Networks. *Mathematics* **2023**, *11*, 2616. https://doi.org/10.3390/math11122616

Academic Editor: Ioannis G. Tsoulos

Received: 28 April 2023
Revised: 3 June 2023
Accepted: 6 June 2023
Published: 7 June 2023

1. Introduction

Cryptographic algorithms are a ubiquitous part of modern life since they allow us to preserve the confidentiality and integrity of sensitive data. However, the implementation of such algorithms (even if they are mathematically secure) can sometimes leak information about security assets, for instance, through power [1] or electromagnetic radiation [2,3]. An attacker can attempt a *side-channel analysis* (SCA) on leakages to exploit that leakage and retrieve the secret key or its parts.

Assuming that the attacker has an identical copy of the target device, *profiling* SCA becomes one of the most potent attack methods. Such an attack leverages traces generated on the copy to construct a model that profiles leakage patterns corresponding to the key-related intermediate data or the key itself. The profiling model can then recover the secret key from traces generated by the target device. Recently, neural networks (NNs) became popular profiling models, since they are able to achieve good attack performance even on devices that are protected by SCA countermeasures [4–8]. These types of attacks are commonly referred to as deep learning-based side-channel analysis (DL-SCA) [9].

The *architecture* of an NN refers to its inner components, such as the neurons and the connections in between. In side-channel analysis research, an NN's architecture is often decided empirically, resulting in different architectures even on the same dataset [4,7,8]. In practice, one of the biggest obstacles to applying DL-SCA is the design of the NN

architecture and the optimization of its hyperparameters. As a matter of fact, a NN can be composed of hundreds of neurons and, therefore, of hyperparameters that specify the NN's architecture. This gives rise to a combinatorial problem: exhaustively testing all hyperparameter combinations to find the best-performing one is unfeasible in practice, given the enormous amounts. Even worse, the selected architecture may not be transferable when attacking different datasets or implementations. Therefore, it would be helpful to have a sophisticated and automated approach to build an architecture for an SCA on any given dataset. While there are other approaches to the automated design for neural networks for SCA, they also come with specific issues. For instance, Rijsdijk et al. used reinforcement learning that produced top-performing neural networks, but the authors still needed to start with a general description of the architectures to be designed [10]. Additionally, a reinforcement learning approach is computationally expensive and requires a cluster of GPUs and tuning time days. On the other hand, Wu et al. used Bayesian optimization to find neural network architectures for SCA [11]. This approach is much faster than reinforcement learning while providing similar results (in terms of attack performance). Still, the authors needed to select the surrogate model and acquisition function for Bayesian optimization, which can again make hyperparameter tuning significantly harder. Moreover, both aforementioned methods rely on the experience obtained from iterations, and the question "*Is the selected model global optimal?*" is tricky to answer.

In this paper, we propose a *genetic algorithm* (GA) as an alternative to the above-mentioned methods for the hyperparameter tuning task in the context of DL-SCA. In general, GAs are quite versatile metaheuristics for hyperparameter tuning. They optimize a population of candidate hyperparameter vectors (or individuals, in GA terminology). A GA mimics the natural evolution of these vectors by applying genetic operators, such as recombination and mutation, pruning the population with a selection method, and evaluating them against a fitness function. This process is iterated over multiple generations, after which, the last generation's best-performing hyperparameter vector is taken as a solution. In principle, a GA allows for obtaining robust models for leakages acquired from different cryptographic implementations.

The main contributions of this work are:

1. We provide a methodology based on genetic algorithms for tuning the hyperparameters of neural networks used in profiling side-channel analysis. Our approach is automated, extensible, and capable of producing various neural networks.
2. We analyze the components of well-performing architectures constructed with our method, giving insights into the effectiveness of CNN hyperparameter options for side-channel analysis.

The rest of this paper is organized as follows. In Section 2, we provide information about profiling SCA, neural networks, genetic algorithms, and the datasets we use. Section 3 provides an overview of related works. In Section 4, we provide details about our novel methodology. Section 5 provides details about the experimental setup and reports the obtained results. In Section 6, we provide a discussion about the obtained architectures. Finally, in Section 7, we conclude the paper.

2. Background

In this section, we cover all necessary background concepts related to side-channel analysis, neural networks, and genetic algorithms that form the basis of our contribution. The treatment is essential, as a complete overview of these subjects is clearly out of the scope of this manuscript. For further information, the reader can refer to Picek et al.'s recent paper about the systematization of knowledge [9].

2.1. Profiling Side-Channel Analysis

When analyzing the security of a system, the first step always requires the definition of a *threat model* that specifies precisely the capabilities of an attacker. This commonly includes the adversarial goal (what the attacker is aiming for), the type and amount of information

about the system that can be exploited, and the computational resources to perform an attack. As we mentioned in the introduction, the aim of SCA is to infer secret information from a cryptographic device (usually, the encryption key) by observing and measuring its physical leakages. The value of the key cannot be directly measured through side-channel analysis since it would assume that an attacker can basically control everything related to the cryptographic device, yielding an unrealistic threat model. Instead, a *chosen plaintext attack* is usually assumed: the adversary has access to a clone device equipped with the secret key meant to be discovered, and they can encrypt whatever plaintext they wish using this key, observing the corresponding ciphertext and any side-channel leakage that occurred during the encryption process. This type of attack is also called a *profiling attack* in the SCA context.

A profiling attack consists of a profiling phase and an attack phase, which are roughly analogous, respectively, to the training and test phases in the context of supervised learning machine learning. In the profiling phase, an attacker uses leakages from a clone device to construct a model that maps the relationship between leakages and corresponding labels (i.e., key-related intermediate data). In the attack phase, the attacker iterates over all possible key candidates and obtains the respective output probabilities for their labels. By repeating this process for each trace and summing the logarithms of the probabilities assigned to each key candidate, the attacker ends up with a log probability vector used to determine the likelihood of each candidate being the correct key. Equation (1) formulates the procedure of obtaining the log probability for a key candidate k' over N attack traces:

$$P_{log}(k') = \sum_{i=0}^{N-1} \log(P(l(p_i, k'))), \tag{1}$$

where p_i denotes the plaintext used for trace i, $l(\cdot)$ is the cryptographic operation producing the targeted intermediate data, or equivalently, the output label, and $P(\cdot)$ is the probability assigned by the profiling model to the output label passed as an argument. Thus, Equation (1) sums the log of the probabilities assigned by the profiling model to each output label obtained by encrypting different plaintexts under the same key candidate k'.

The attack performance is evaluated with the *key rank* metric [9] as follows:

$$\mathsf{KR} = \left|\{k' | P_{\log}(k') > P_{\log}(k)\}\right|. \tag{2}$$

Intuitively, the key rank is the number of key candidates with a higher likelihood of correctness than the correct key value. An attack is successful when the correct key is predicted with the highest likelihood or can be brute-forced after being placed among the few highest-likelihood candidates. In this work, we discuss the mean key rank achieved over multiple experimental runs, in which case, the metric is commonly referred to as the *guessing entropy* [12]. Furthermore, as common in the related works, we will assess the attack performance against a single key byte only (which is denoted as partial guessing entropy), but for simplicity, we will denote it as guessing entropy. A common assumption is that attacking a single key byte reveals the average effort required for other key bytes as well [10,13].

2.1.1. Countermeasures

SCA countermeasures aim to mitigate the information leakage produced during cryptographic operations, and they can be classified into two main groups: *masking* and *hiding* countermeasures [9]. In the masking approach, the sensitive intermediate values are split into different shares to decrease the key dependency [3]. In a hiding countermeasure, the goal is to make the traces look similar to random noise. Usual approaches to design hiding countermeasures include random delay interrupts and desynchronization techniques. In this work, we focus on Boolean masking and desynchronization, since they are the most commonly adopted SCA countermeasures.

For instance, as implemented in the considered datasets [4], a random mask r is applied after the AddRoundKey operation in the first round of AES encryption. The intermediate value is then computed as:

$$Z = \mathsf{SBOX}(p \oplus k) \oplus r. \tag{3}$$

Desynchronization is a common type of hiding countermeasure that introduces time randomness to the leakages. In practice, such effects can be realized by adding clock jitters and inserting random instructions. We simulate this effect by randomly shifting with an upper bound for each trace [4,6].

2.2. Neural Networks

SCA can be considered a classification task that aims to map the input leakages to a cluster corresponding to the targeted labels. Such a task can be accomplished with a *neural network* (NN), which is essentially a nonlinear function composed of layers of *neurons*, sometimes referred to as *nodes*. The output of a neuron is defined as follows:

$$y = \phi(\sum_{i=i}^{n} w_i x_i + b_k), \tag{4}$$

which is computed by multiplying the neuron's inputs $x_1, \ldots, x_n$ from the previous layer with their corresponding weights $w_1, \ldots, w_n$, adding the *bias* value b_k corresponding to neuron k, and transforming the result with the *activation function* ϕ. The activation function acts as a source of nonlinearity and often improves the efficiency of the training phase. Two common activation functions for NNs in SCA are the *rectified linear unit* (ReLU) and *scaled exponential linear unit* (SELU).

When training a neural network, the weight and bias of each neuron are updated with gradient descent to minimize the loss function. *Categorical cross-entropy* (CCE) is one common loss function in multi-class classification problems. Cross-entropy is a measure of the difference between two distributions. Minimizing the cross-entropy between the true distribution of the classes and the distribution modeled by the neural network improves its predictions:

$$CCE(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{l} \sum_{j=1}^{c} y_{i,j} \cdot \log(\widehat{y_{i,j}}), \tag{5}$$

where c and l, respectively, denote the number of classes and data, y is the true value, and $\hat{y}$ is the predicted value.

A primary type of NN architecture in SCA is the *multilayer perceptron* (MLP), in which a sequence of fully connected hidden layers of neurons is followed by an output layer that transforms the final output values to label prediction probabilities. A *convolutional neural network* (CNN) is another commonly used type of network in SCA. It prepends its first fully connected layer with one or more convolutional blocks. Such a block consists of a *convolutional layer* that attempts to compute local features over the input data, and it is optionally followed by a *pooling layer* that aggregates the resulting values, e.g., by calculating n-wise averages. Equation (6) formally displays the application of j convolution filters with the kernel size k on inputs $(x_i, x_{i+1}, \ldots, x_{i+j})$. A convolutional layer repeats such convolutions until it shifts through all n inputs, resulting in $n \cdot j$ inputs for the fully connected layers. Formally, this operation can be stated as follows:

$$\begin{aligned} \mathsf{conv}((x_i, x_{i+1}, \ldots, x_{i+j}\}) = \{&c_{0,0}c_{0,1} \ldots c_{0,k-1}x_i, \\ &c_{1,0}c_{1,1} \ldots c_{1,k-1}x_{i+1}, \\ &\ldots, \\ &c_{j-1,0}c_{j-1,1} \ldots c_{j-1,k-1}x_{i+j}), \end{aligned} \tag{6}$$

where the $c_{l,m}$ for $0 \leq l < j$ and $0 \leq m < k$ represent the coefficients of the j convolution filters.

2.3. Genetic Algorithms

A genetic algorithm (GA) is a type of population-based optimization algorithm that typically utilizes elements from biological evolution [14]. A GA's objective is to optimize a solution to some problem by maintaining a population of such solutions and evolving them over several *generations*. We refer to such a solution as an *individual* or *genome* consisting of building blocks known as *genes*. One generation is formed by evaluating the fitness of each genome, selecting fit genomes as parents for reproduction, and applying genetic operators, such as mutations and crossovers on those parents to generate the offspring, which represents the next generation.

Before commencing the first generation, the genomes in the population are randomly initialized for diversity. One then starts an iteration of generations until the fitness evaluation budget expires or the fitness value of the best genome achieves a predefined threshold. Each generation starts with a *fitness evaluation*, by assigning a fitness value to each genome that measures how well the corresponding individual performs concerning the relative optimization problem. The next step is *selection*, which aims to cull weak genomes from the population so that the algorithm favors genetic modifications that create fitter genomes. Rather than straightforwardly selecting a number of the fittest genomes, modern GAs employ more sophisticated methods to preserve diversity in the population. One such method is *tournament selection* [15–17], which determines parents by holding 'tournaments' of some randomly picked genomes and retaining those with the best fitness as parents.

After selecting the parents, a GA typically generates as many offspring as the number of parents. The production of one child's genome involves applying one or multiple *genetic operators* to one or multiple parents. These operators include *mutations* and *crossovers* [16–18], although the latter can be omitted in a non-mating GA. Mutation only requires one parent, which is cloned and randomly modified with a predefined mutation function to produce one child. On the other hand, crossover refers to the combination of properties of two or more parents to construct a child. In this work, we apply the *polynomial mutation*, a mutation method for real-valued parameters originally introduced by Deb and Agrawal for GA in [19], and then apply it to other metaheuristics, such as multi-objective evolutionary algorithms [20] and differential evolution [21]. This mutation operator is designed for variables with predefined minimum and maximum boundaries. The method mutates a variable x towards the lower boundary x_L or the upper boundary x_U with a uniform probability. The degree of the mutation is then determined by a pseudorandom number $0 \leq u < 1$ and parameter η, with a higher value of η resulting in a smaller mutation range. In other words, the mutated value x' is equal to $x + \bar{\delta}_L(x - x_L)$ or $x + \bar{\delta}_R(x_R - x)$ with $\bar{\delta}_L$ and $\bar{\delta}_R$ scaling with u and η, as defined in Equation (7).

$$\begin{aligned} \bar{\delta}_L &= (2u)^{\frac{1}{1+\eta}} - 1 \\ \bar{\delta}_R &= 1 - (2(1-u))^{\frac{1}{1+\eta}}. \end{aligned} \tag{7}$$

Neuroevolution refers to the use of an evolutionary algorithm for constructing or optimizing an NN. In this work, we will use a GA to construct a neural network architecture for SCA. In such a scenario, a genome in a GA describes the hyperparameter combination of an NN. The fitness is determined through the evaluation of the network's performance.

2.4. Datasets

We used the ANSSI SCA Database (ASCAD) [4], where each trace comprises 700 trace points corresponding to the S-box operation of the third key byte. This is a standard dataset in the SCA research community, which is used as a common benchmark to evaluate the performance of a particular SCA attack or countermeasure technique [7,8,10,11]. For this reason, we decided to adopt it for our experiments and compare our results with those achieved by other state-of-the-art methods. Note that we are referring to the *fixed-key* ASCAD dataset, where the same encryption key is used in all AES operations. These traces

are protected with the *masking* countermeasure, so the intermediate value Z was computed as in Equation (3) for a random mask byte r.

The training (35,584) and validation (3840) sets were balanced samples taken from the 50,000 training traces. Their respective numbers were chosen such that both sets were sufficiently large for their respective purposes. Since we used the identity leakage model in all our experiments with this method, the numbers of both sets are multiples of 256, which corresponds to the number of output labels. Note that while we do not expect many issues with the identity leakage model and class imbalance [22], we still balance the classes to mitigate any undesired effects. Ten thousand attack traces were used to assess the attack performance. Finally, we conducted experiments on this dataset with and without the desynchronization countermeasure.

3. Related Work

We now provide an overview of the literature concerning optimizing NN's architectures for the SCA domain, considering both manual and fully automated approaches. Next, we will briefly survey the automated methods based on neuroevolution.

3.1. Network Architecture Optimization in SCA

To the best of our knowledge, the first work concerning the use of deep learning for SCA dates back to Maghrebi et al [23]. There, the authors investigated the effectiveness of several SCA attacks enhanced by DL on the DPAv2 dataset [24] and custom AES implementations with and without first-order masking. The neural networks obtained through this method achieved a null key rank by using less than 10^3 training traces on the masked implementation. Interestingly, the CNN architecture was determined with a genetic algorithm by using the guessing entropy as a fitness function, but the authors did not provide a detailed elaboration on their methodology.

Benadjila et al. [4] further explored the performances of neural networks in the SCA context. In particular, the authors observed that the VGG-16 model proposed in [25], or CNNs with a similar architecture, achieved good results for SCA. Next, Kim et al. [5] employed the data augmentation technique to improve the performance of the SCA attack based on a CNN. This method was found to be effective at improving the CNN when dealing with different types of SCA countermeasures. Successively, Zaid et al. introduced new architectures for CNNs in [7], which were shown to achieve state-of-the-art performances while significantly reducing the sizes of the neural networks. Wouters et al. [8] further reduced the network size with data preprocessing strategies.

In addition to manual optimization of neural networks, recent research has also explored fully automated approaches for network architecture searches. Rijsdijk et al. [10] customized the MetaQNN reinforcement learning algorithm for SCA to automatically find CNN architectures. However, the search space is roughly limited to hyperparameters that we know to be effective, and pure MLP architectures are not discussed. Each NN is evaluated by training it for 50 epochs using the Adam optimizer and the SELU activation function in their work. Wu et al. proposed AutoSCA [11], which uses the *Bayesian optimization* to find architectural hyperparameters for both MLPs and CNNs. Their approach produced good results and mainly focused on finding larger architectures with at least 100 neurons in each dense layer.

3.2. Evolution-Based Network Architecture Search

Evolutionary approaches have been widely used in automated network architecture searches. Real et al. [26] developed one such method for image classification on modern datasets, a task that requires large networks. They propose a non-mating GA with both NEAT-like [27] mutations and layer-level mutations to evolve CNNs on granular and large scales. Specifically, each genome is trained with backpropagation on 45,000 samples before evaluating its fitness. In this approach, a child's genome keeps the weights and biases of its parent, effectively training each network over time.

Other successful neuroevolution methods that construct CNNs for image classification include the deep evolutionary network structured representation approach, DENSER [28], and EvoCNN [29]. DENSER uses a two-level genotype, where the first level encodes the NN hyperparameters while the second encodes layer-specific variables, such as the number of neurons or the variables of a convolutional filter. This structure enables the algorithm to be used for MLPs, CNNs, and other types, as long as they can be appropriately defined in the genome's second level. Furthermore, DENSER trains NNs with backpropagation before evaluating the fitness on a validation set. EvoCNN works similarly but evolves the weight initialization values along with the architecture's hyperparameters.

3.3. Neuroevolution for SCA

The use of neuroevolution to perform side-channel analysis has scarcely been explored in existing work. Knezevic et al. used genetic programming to evolve custom activation functions specifically designed for side-channel analysis [30]; these functions can outperform the widely used ReLU function. In their approach, the genome encodes an activation function as a tree structure consisting of unary and binary operators, with leaves representing the function's inputs. Such a tree is initially initialized with a depth ranging from two to five levels and is limited to twelve levels during evolution. For the fitness evaluation, they computed the mean number of attack traces required to obtain a key rank of zero over one hundredfold, and then add it to one minus the accuracy. The method resulted in novel activation functions that improved the performances of large and efficient MLPs and CNNs. Acharya et al. proposed InfoNEAT [31], an approach that tailors the *neuroevolution of augmenting topologies* (NEAT) algorithm [27], specifically for the side-channel analysis. Their approach considers the identity leakage model and uses NEAT to evolve an NN architecture with a single output node for each of the 256 output classes. The resulting 256 binary networks are combined by a *stacking* approach that uses the network outputs as inputs for a logistic regression model. Such a stacked model is created for multiple folds of balanced trances taken from the complete dataset, after which, those model prediction probabilities can be summed to form a final prediction for each attack trace.

In our preliminary investigations, we also considered the NEAT approach, which focused on evolving only the architecture of a NN for SCA attacks, while InfoNEAT targeted both the architecture and the weights of the network. However, the results on the ASCAD dataset were definitely not encouraging and, thus, we dropped NEAT to develop our own NASCTY methodology. A full account of the experiments with the modified InfoNEAT algorithm on the ASCAD dataset can be found in the first author's thesis [32].

4. NASCTY

Neuroevolution to attack side-channel traces yielding convolutional neural networks (NASCTY-CNNs) is a GA that modifies hyperparameters of CNNs for the side-channel analysis. Algorithm 1 shows the main procedure of our approach. This section will specify the genome structure, the initialization of the population, the fitness evaluation method, and the method used to produce offspring.

Algorithm 1 The NASCTY-CNNs algorithm.

1: $train\_traces, train\_labels, valid\_traces, valid\_labels \leftarrow \mathsf{sample}(ascad\_data)$
2: $pop \leftarrow \mathsf{initialise\_population}()$
3: **while** $gen < max\_gens$ **do**
4: $\quad \mathsf{evaluate\_fitness\_values}(pop, train\_traces, train\_labels, train_plaintexts)$
5: $\quad parents \leftarrow \mathsf{tournament\_selection}(pop)$
6: $\quad offspring \leftarrow \mathsf{produce\_offspring}(parents)$
7: $\quad pop \leftarrow parents \cup offspring$
8: **end while**
9: **return** genome in pop with the lowest fitness

Note that the sampling of training and validation data was only performed once, meaning that we used the same data for the fitness evaluation in every generation. Moreover, we ensured the usage of balanced data samples, wherein a selected set of traces consisted of an equal number of traces associated with each feasible output label. In line with previous works, such as [26,28,29], our selection process involved a tournament size of 3. This indicates that we randomly selected three individuals and identified the most fit individual among them as a potential parent for reproduction. It is important to highlight that all experiments conducted in this study focus on targeting the third (masked) key byte of the fixed-key ASCAD dataset. Each trace in the dataset consists of 700 points, and each point is normalized to fall within the range of -1 to 1, as conducted by Wouters et al. [8].

4.1. Genome Structure

The NASCTY genome represents a CNN and consists of a list that can contain zero to five convolutional blocks, an optional pooling layer when no convolutional blocks are present, and a list of one to five dense layers. Each convolutional block is described with the number of convolutional filters, the filter size, a Boolean denoting the presence of a batch normalization layer, and a pooling layer. Any pooling layer in the genome comprises a pooling type, either max pooling or average pooling, a pool size, and a pool stride. Finally, a dense layer is described only by its number of neurons. An example of the genome structure is presented in Figure 1.

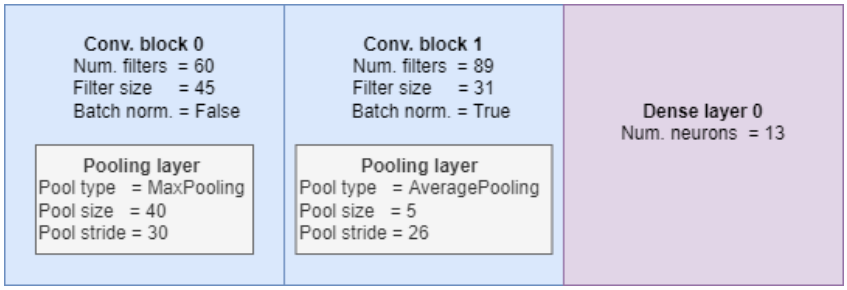


Figure 1. An example of the genome encoding used in the NASCTY algorithm.

When expressing a NASCTY genome as a neural network, following the state-of-the-art architectures, we always use the SELU activation function for all hidden neurons, the He weight initialization for the convolutional blocks and dense layers, and the Glorot uniform weight initialization for the output layer, which uses the softmax activation function. Although enlarging the genome parameter spaces would increase the diversity of the populations, applying the prior knowledge would speed up the evolution process.

4.2. Population Initialization

We initialize the networks by assigning random values to the hyperparameters within the specified ranges detailed in Table 1. Such ranges and the structure of the genome are inspired by previous research on SCA, particularly the VGG-like networks mentioned in [4,5]. Additionally, we incorporate ideas from the reinforcement learning-based approach for the automated architecture search in SCA described in [10].

Notably, to reduce the time required for the evaluation, one can initialize the population with architectures that have a minimal number of trainable parameters. However, we opt to completely initialize the population at random to avoid local optima that may come about due to the reduced diversity in the population.

Table 1. Ranges for CNN genome hyperparameters in the NASCTY algorithm.

Parameter	Options
Num. convolutional blocks	0 to 5 in a step of 1
Num. dense layers	1 to 5 in a step of 1
Num. convolutional filters	2 to 128 in a step of 1
Filter size	1 to 50 in a step of 1
Batch normalization layer	False, True
Pooling type	Average, Max
Pool size	2 to 50 in a step of 1
Pool stride	2 to 50 in a step of 1
Num. dense neurons	1 to 20 in a step of 1

4.3. Fitness Evaluation

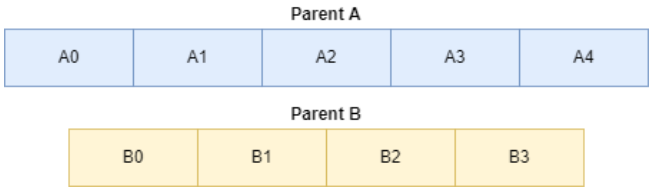
Once a genome is defined, the corresponding CNN is trained using the Adam optimizer (all networks are trained with the same seed in every generation to ensure a fair comparison between them). The loss value on the validation set is used for the fitness evaluation. By minimizing loss, we aim to have a system aligned with the related works in DL-SCA. Naturally, one could consider other options here, for instance, the ones applied in [11]. The purpose of training the networks prior to evaluation is to enhance our ability to discern their quality with greater precision. Following previous papers on the subject [11,28,29], we opted to train each network for ten epochs. Our initial experiments, conducted according to the methodology recommendations provided by [29], demonstrated that this duration is sufficient for networks of similar sizes to exhibit noticeable differences in their CCE values, indicating varying levels of quality.

The objective of training the networks before evaluating them is to enable us to differentiate their quality more accurately. We chose to train each network for ten epochs as other works do [11,28,29] and preliminary experiments following the methodology recommendations by [29] showed that this is enough for similarly sized networks to observe significant CCE differences in networks of different qualities.

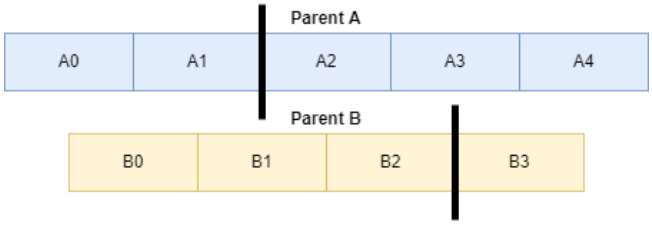
4.4. Offspring Production

After evaluating the fitness of each genome in one generation, the members of the next generation (offspring) can be produced. Half of these members are produced by applying the tournament selection to the population to find fit genomes that will act as parents. The remaining half, on the other hand, is constructed by randomly choosing pairs of those parents on which the crossover and mutation operations are applied. Optionally, we may only apply tournament selection to a proportion of the top-performing members of the population to select parents. This operation is performed to ensure that the best genomes are maintained in the population, a concept known as elitism, which can be tuned with a *truncation proportion* parameter.

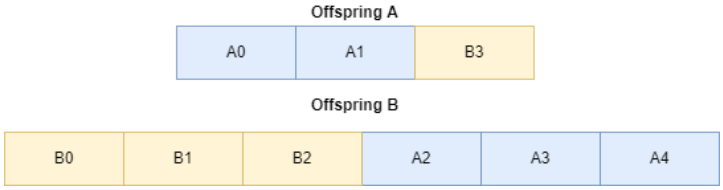
Our algorithm uses one of two possible crossover types, i.e., *one-point* crossover or *parameter-wise* crossover; both are common crossover strategies in genetic algorithms. The performances of these two methods are evaluated in Section 5.2. To enact a one-point crossover with two parents, we apply one-point crossover separately on the parents' lists of convolutional blocks and dense layers. For a list of either convolutional blocks or dense layers, we achieve this operation by picking a random cutoff point in both parents' lists of the corresponding layer type. The first child's list of that type is then created by connecting the first parent's list before its cutoff point to the second parent's list after its cutoff point; the second child's list is created by connecting the remaining units. The one-point crossover operation is then finalized by randomly dividing the parents' optional pooling layers that are present in the absence of convolutional blocks among the offspring (see Figure 2).



(**a**) Each parent's list of layers or blocks.



(**b**) The parents' respective random cutoff points.



(**c**) The two lists of layers or blocks present in the offspring after the crossover.

Figure 2. Visualization of the one-point crossover on a list of convolutional blocks or dense layers in a NASCTY genome.

In our implementation of the parameter-wise crossover, the first child's genome is created by iterating over pairs of convolutional blocks from the parents and randomly inheriting convolutional block genes from either parent. The second child's genome then inherits the remaining hyperparameters for these blocks. The exact process is repeated for the parents' lists of dense layers. In the typical scenario where one parent has more convolutional blocks or dense layers than the other parent, the excess units are appended to the first child's genome without any modifications.

After the crossover operation, the offspring are mutated through one of the following methods with a uniform probability:

- Adding one random convolutional block or dense layer with randomly initialized hyperparameters;
- Removing one random convolutional block or dense layer;
- Modifying all hyperparameters through *polynomial mutation* with a probability of $\frac{1}{n}$, where n is the total number of modifiable hyperparameters in the genome.

The polynomial mutation is described in Section 2.3 and is designed for variables with predefined minimum and maximum boundaries, which fits our task of exploring proper hyperparameter values within predefined ranges. This genetic operator is defined by a parameter η, also known as the distribution index in [33], which controls the similarity between the child solutions and their parents. A large value for η implies that the mutated children will be very close to their parents; conversely, a small value will yield children that differ more from their parents. The related literature recommends a range for η between 20 and 100. Of course, an appropriate value for this parameter can be determined more precisely by tuning it experimentally on the specific optimization problem.

5. Experiments

In this section, we discuss the experimental evaluation of NASCTY. We start by describing the setup of our experiments. Then, we show the outcome of the preliminary

tuning phase based on a grid search. Finally, we present the results obtained by the GA with the best-performing parameter combination on masked and desynchronized traces of ASCAD.

5.1. Experimental Setup

For the experimental validation of our approach, we optimize GA parameters through a grid search, then evaluate the performance on the masked ASCAD traces and masked and desynchronized ASCAD traces for several desynchronization levels. The objectives of these experiments are to determine:

- The effectiveness of GA parameters for our approach;
- Whether our automated approach can produce NNs that outperform similar NNs found through trial and error;
- Architecture components that contribute to the effectiveness of an SCA.

To account for the randomness introduced by the mutation operations, we ran five experiments for each GA parameter configuration and report the best results. The best genome resulting from the NASCTY algorithm was evaluated by training the corresponding NN for 50 epochs and computing the mean key rank, which is equivalent to the guessing entropy [12], over 100 folds.

All experiments were executed with 52 parallel workers, each of which ran at approximately 2.1 GHz on an Intel E5-2683 v4 CPU. With these computational resources, the discussed experiments required 84 GB RAM and took at least four days, but no more than seven days, to complete. This significant variance in runtime complexity is caused by the pseudorandom nature of GAs, which results in the construction and evaluation of NNs of varying sizes.

5.2. Parameter Tuning by Grid Search

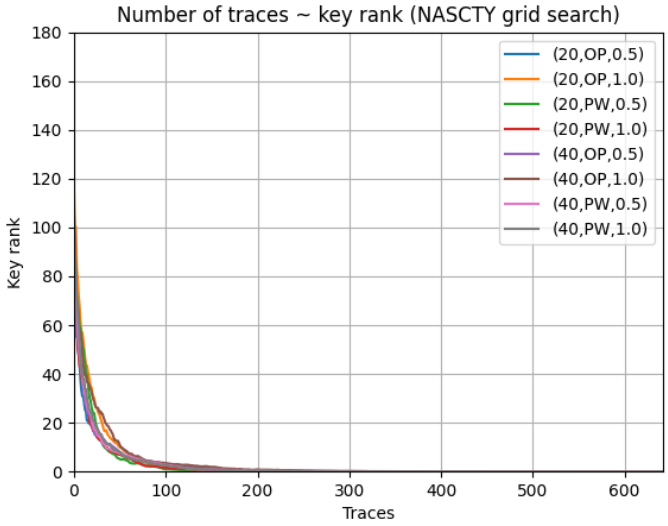
Table 2 shows a summary of the GA parameters for the grid search. Note that additional parameter options, mutation strategies, and crossover strategies could potentially result in better performance, but such adjustments would have to significantly diverge from our current strategy to assess the general effectiveness of the algorithm. We ran each grid search experiment with a population size of 52 to match the number of available parallel workers and ran the GA for ten generations. Furthermore, each experiment used the same training data, validation data, and initial population to observe the impact of the parameter changes more accurately.

Table 2. The parameter values we considered during the grid search for NASCTY.

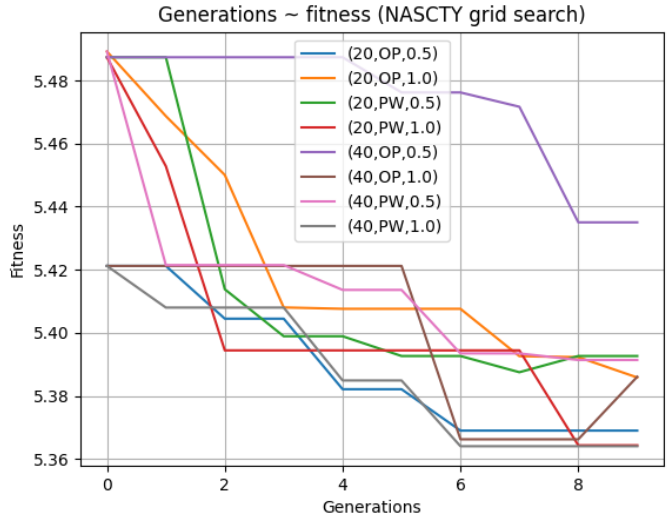
Parameter	Options
Polynomial mutation η	20, 40
Crossover type	One-point, parameter-wise
Truncation proportion	0.5, 1.0

We expect that setting the distribution index η for the polynomial mutation operator [19] is more suitable for our experiments, where we run the GA for relatively few generations. As we mentioned before in Section 4.4, smaller values of η cause the mutated children to be less similar to their parents. Hence, using a small distribution index could be beneficial in terms of finding better-performing networks in fewer generations. However, one also needs to strike a balance to avoid potentially damaging mutations by choosing a value that is too small [33]. For this reason, we settled for $\eta = 20$. Similarly, a larger truncation proportion would yield better results since, in this way, the diversity of the population (which is already limited in size) is preserved. However, its influence is likely not as significant as that of the chosen crossover and mutation configurations since those can modify the population more straightforwardly. Finally, we expect either crossover

strategy to perform well since both allow the algorithm to find effective architectures in the predefined hyperparameter ranges. The performance of the final network obtained from the best run for each parameter combination is depicted in Figure 3.



(**a**) The mean performance of the best runs over 100 folds.



(**b**) The fitness progressions of the best runs (the lower the better).

Figure 3. NASCTY grid search results corresponding to the best network obtained with each parameter combination.

Figure 3a implies that each parameter combination is capable of producing fit architectures for the considered ASCAD traces. Similarly, the fitness plots of the best runs over generations shown in Figure 3b demonstrate that the best genome's CCE, i.e., the validation loss, can improve significantly in as few as ten generations, regardless of the parameter combination under consideration.

The best mean incremental key rank among all grid search experiments was approximately 0.50419 and resulted from the experiments with a polynomial mutation η value of 20, a 1-point crossover, and a truncation proportion of 1.0. Therefore, those parameters are applied for all further experiments. To determine each GA parameter's influence on the final performance, we observe the effect of modifying one variable at a time while keeping the others constant at the aforementioned best-observed values. Table 3 displays how such modifications affect the mean incremental key rank. From the table, we can infer that only the crossover strategy significantly affects the final performance among

the parameters we considered; one-point crossover is preferred over the parameter-wise crossover. Since the best runs using parameter-wise crossover in Figure 3 still perform well, the performance difference likely results from poor consistency compared to runs using the one-point crossover. We suspect that the additional consistency observed with the one-point crossover is achieved through its advantage in retaining functional sequences of convolutional blocks or dense layers. In addition, one-point crossover on lists of layers intuitively provides synergy with our mutation strategy of adding or removing an entire layer because effective additions or removals can be identified more quickly when they are separated into offspring in a modular fashion.

Table 3. The effect of each parameter on the final incremental key rank in NASCTY grid search experiments.

η	Crossover Type	Truncation Proportion	Mean Incremental Key Rank
20	One-point	1.0	0.50419
40	One-point	1.0	0.50880
20	Parameter-wise	1.0	0.89354
20	One-point	0.5	0.50966

5.3. ASCAD: Masked and Desynchronized

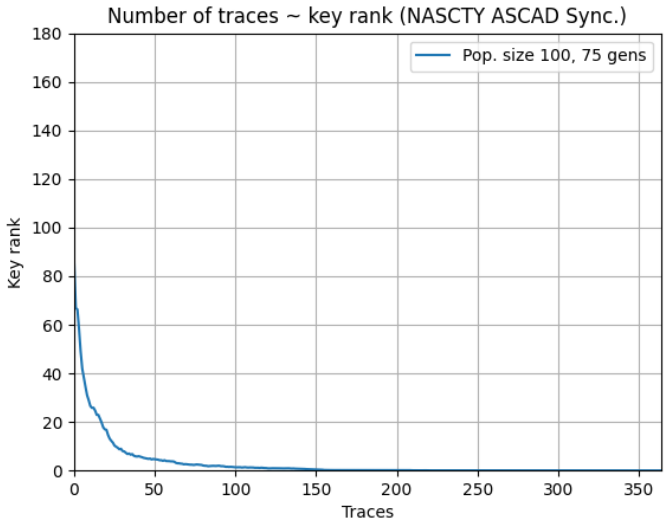
All remaining experiments were run with the best-performing parameter options found through our grid search experiments. In addition, we ran these experiments with a population size of 100 to fully exploit the resources at our disposal. In contrast to our grid search experiments, these experiments did not use a seed for the pseudorandom numbers involved anywhere in the GA except for the fitness evaluation procedure, where we used a seed for the training of each NN to ensure that the genomes were fairly compared.

We first ran NASCTY on masked ASCAD traces for 75 generations to evaluate the algorithm's general effectiveness. Then, we ran experiments on the same masked dataset, which was further protected with desynchronization as described in Section 2.1.1. Specifically, we ran 3 sets of experiments with desynchronization levels of 10, 30, and 50, respectively. With this approach, we aim to determine whether NASCTY can circumvent or mitigate countermeasures without additional algorithm modifications and whether larger desynchronization levels hinder NASCTY's ability to find good architectures.

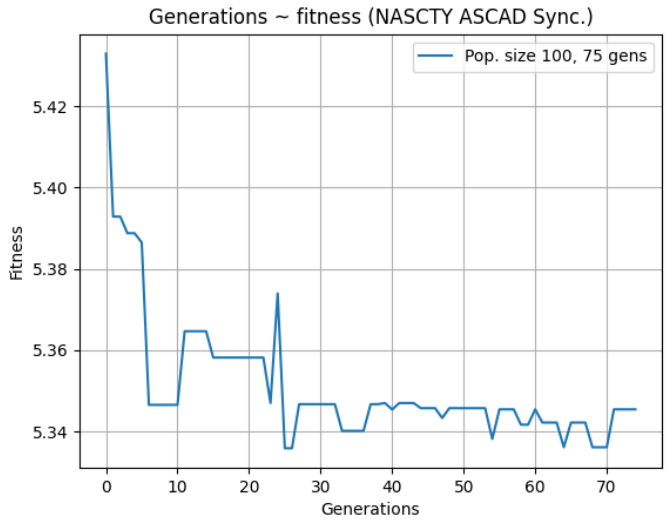
The fitness progression trends in Figure 3b indicate that more generations would improve the observed fitness of the best genome. Following this, we first ran NASCTY on masked ASCAD traces for 75 generations with a population size of 100. Figure 4 shows the results of this first experiment.

As shown in Figure 4a, the best-obtained network converges smoothly. Ultimately, the network breaks the target into 314 attack traces and achieves a mean incremental key rank of 0.51857. The best fitness value progresses (Figure 4b) continually after ten generations have passed, then stagnates well before the seventy-fifth generation is reached. Moreover, despite the difference in the population size and the number of generations, the best network is outperformed by several of the NNs obtained with our grid search experiments. This observed fitness stagnation implies that the algorithm may be prone to becoming stuck in local optima. Typically, the mutation is the source of the global search in a GA, so we recommend that future work evaluates lower values of η for polynomial mutation strategies or possibly more perturbing mutation strategies.

Due to the fitness progress observation in the previous experiment, we ran NASCTY on the masked and desynchronized ASCAD traces for 50 generations instead of 75. The results for desynchronization levels 10, 30, and 50 are displayed in Figure 5.



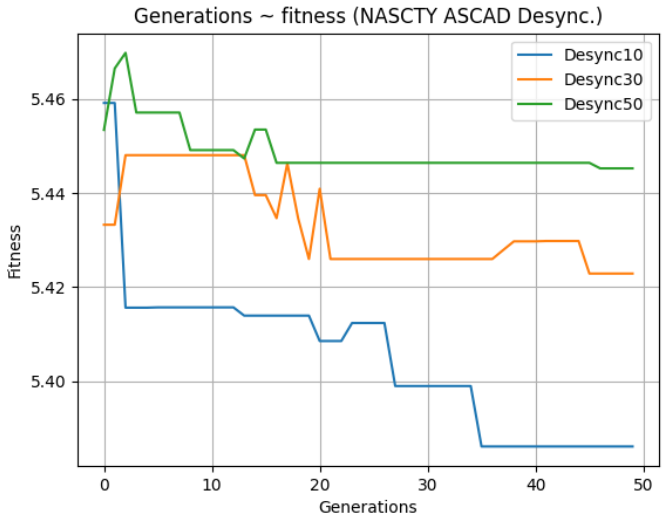
(**a**) The mean performance of the best neural network over 100 folds.



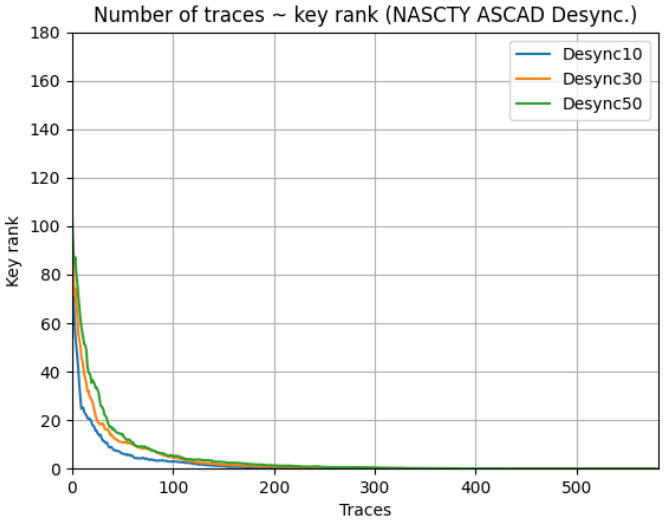
(**b**) The fitness progression of the best run.

Figure 4. NASCTY results corresponding to the best network obtained on masked ASCAD traces.

As shown in Figure 5a, the time-randomness introduced by desynchronization affects the algorithm's performance, considering that both fitness progress and final performance are noticeably diminished as the desynchronization level increases. Still, the results show that NASCTY can find effective architectures despite the added countermeasures, with the networks evaluated in Figure 5b being able to obtain key rank 0 in 338, 474, or 531 traces, respectively, for desynchronization levels 10, 30, and 50.



(**a**) The fitness progressions of the best runs.



(**b**) The mean performance of the best runs over 100 folds.

Figure 5. NASCTY results for ASCAD traces at desynchronization levels 10, 30, and 50.

6. Discussion

The best run on the synchronized ASCAD traces produced the CNN architecture shown in Figure 6a. It has 10,470 trainable parameters and vaguely resembles the efficient CNN proposed by Zaid et al. [7].

In comparison, the architecture produced with NASCTY has an additional dense layer and possesses several unintuitive components, such as 27 convolutional filters of size 45 and a pool stride that exceeds the pool size. Since the efficient MLP proposed by Wouters et al. [8] only required two layers of ten neurons each, we surmise that NASCTY may be inclined to include the unnecessary model complexity. In other words, NASCTY does not sufficiently discourage redundant model complexity. Indeed, by increasing the desynchronization from 30 to 50, the number of trainable parameters of the best architecture decreases from 90,379 to 68,427. A possible solution would be to introduce a model size penalty in the fitness function. Still, NASCTY, with the current configuration, is sufficient at generating good network architectures. Tables 4 and 5 show how NASCTY compares with other state-of-the-art automated hyperparameter tuning methods for SCA, respectively, for synchronized traces and a desynchronization level of 50.

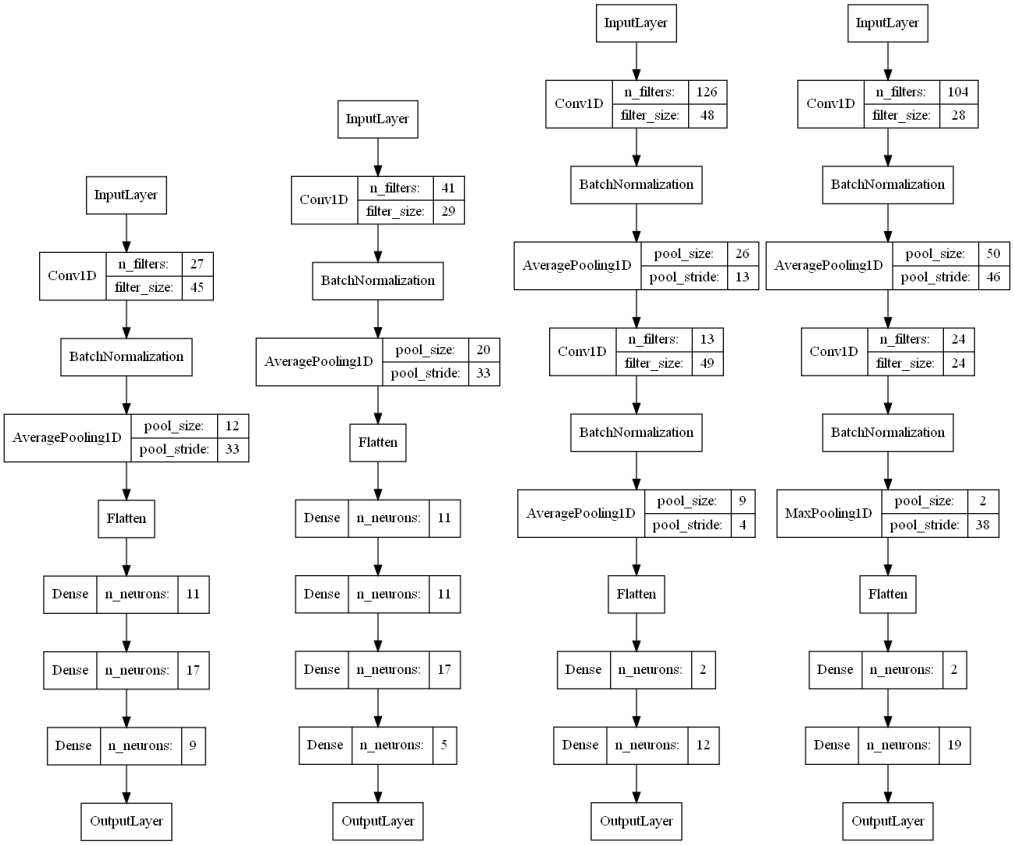


Figure 6. Best architectures produced by NASCTY for masked ASCAD traces protected with several desynchronization levels. (**a**) Synchronized; (**b**) desynchronization level 10; (**c**) desynchronization level 30; (**d**) desynchronization level 50.

Table 4. Comparison of automated hyperparameter tuning methods for SCA on synchronized, masked ASCAD traces.

Method	Num. Traces to Obtain Mean Key Rank 0	Num. Trainable Parameters
Zaid et al. [7]	191	16,960
Wouters et al. [8]	≈200	6436
RL-SCA [10]	242	1282
AutoSCA [11]	158	54,752
NASCTY	314	10,470

Table 5. Comparison of automated hyperparameter tuning methods for SCA on masked ASCAD traces with a desynchronization level of 50.

Method	Num. Traces to Obtain Mean Key Rank 0	Num. Trainable Parameters
Zaid et al. [7]	244	87,279
Wouters et al. [8]	≈250	41,052
RL-SCA [10]	242	1282
NASCTY	531	68,427

It can be seen that the performances are similar to the state-of-the-art methods proposed in [7,8,10,11]. NASCTY requires a few hundred traces to achieve a mean key rank of 0, and the number of trainable parameters ranges in the order of tens of thousands. It can also be observed that NASCTY yields slightly suboptimal results in terms of the number of traces compared to the other methods, but we deem that by further optimizing the GA hyperparameters, it could decrease up to the point of reaching the same level as the other methods, or even outperform them. We leave these experiments as a direction for future research.

The architecture corresponding to a desynchronization level of 10 is similar to the architecture for synchronized traces in both structure and size, with the main difference being the addition of more convolutional filters and another dense layer. Both architectures also feature a pooling layer with a stride that exceeds its size, suggesting their convolutional layers have produced features that are either redundant or incorrectly utilized.

The NASCTY architectures for the two more severe desynchronization levels (30 and 50) provide more insight into NASCTY's way of mitigating this countermeasure. As can be seen in Figure 6c,d, both of these architectures start with a convolutional layer with over 100 filters, a significantly larger number than that of the other architectures' convolutional layers. In addition, both feature two convolutional blocks and two dense layers, with the first dense layer in each network having two neurons. The larger number of filters is consistent with existing approaches to mitigate desynchronization, but the usage of such small dense layers is uncommon when attempting to break protected ASCAD traces. Finally, average pooling appears to be the preferred pooling type in these networks, with max pooling only occurring once.

7. Conclusions and Future Work

This paper proposes a genetic algorithm for the network architecture search in the SCA domain. In NASCTY, each genome encodes the hyperparameters representing a CNN's architecture, and a genome's fitness is evaluated by the validation loss. During offspring production, we apply either one-point crossover on the parents' lists of layers or parameter-wise crossover to create a pair of child genomes that we immediately mutate by adding a layer, removing a layer, or applying polynomial mutation on their genes. With this approach, NASCTY could produce comparable architectures to state-of-the-art techniques. The redundant complexity and unintuitive architecture components found in some NASCTY networks suggest that our method can likely be improved further, implying unexplored potential to match or surpass current state-of-the-art approaches.

Furthermore, NASCTY found effective architectures for traces protected with masking and desynchronization levels up to 50, while keeping its GA parameters and implementation largely unmodified. However, the desynchronization affected the final network performances: networks produced by NASCTY for desynchronization levels 0, 10, 30, and 50 obtained key rank 0 within 314, 338, 474, and 531 attack traces, respectively. We recommend that future work evaluate NASCTY's effectiveness and resulting architecture patterns on traces protected with other countermeasures. The observed network architectures showed that NASCTY tends to combat desynchronization by adding a convolutional layer and increasing the number of filters in the first convolutional layer, which was the case in architectures generated for desynchronization levels 30 and 50. Interestingly, these networks initiated their fully connected part with a dense layer consisting of two neurons. Additionally, some architectures contained pooling layers with a stride larger than their size, which is uncommon in other approaches, and suggests that NASCTY may be generating redundant features through unnecessarily large numbers of convolutional filters. The architectures NASCTY generated for synchronized and mildly desynchronized traces also came with more trainable parameters than models from related work, which achieved better performances on the same task [8], corroborating the hypothesis that NASCTY is prone to adding redundant model complexity. Regardless of the presence of desynchronization, average pooling was preferred to max-pooling in nearly all pooling layers.

In future work, it would be interesting to explore the application of a complexity penalty to the fitness evaluation or population initialization, to reduce the network size by promoting minimal architectures. Furthermore, we recommend experimenting with more mutation parameters to stimulate a better global search to avoid becoming stuck in a local optimum, which often seems to occur well before the algorithm terminates. Once those drawbacks are resolved, we may find better, more interesting architectures, which could possibly outperform the number of traces required by other state-of-the-art methods (although NASCTY comes quite close to them). To do so, we suggest introducing new hyperparameters to the genome, e.g., activation functions for each layer and the learning rate for more general applications. Moreover, one could consider using other types of evolutionary algorithms to perform the search, such as genetic programming, or other metaheuristics, such as swarm intelligence algorithms. Finally, our approach considers the ASCAD dataset with a fixed key. It will be interesting to see how well our approach works with other, more difficult datasets, such as ASCAD with random keys.

Author Contributions: Conceptualization, F.S., L.W. and L.M.; methodology, F.S., L.W. and L.M.; software, F.S.; validation, F.S., L.W. and L.M.; formal analysis, F.S., L.W. and L.M.; writing—original draft preparation, F.S.; writing—review and editing, F.S., L.W. and L.M.; supervision, L.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The NASCTY source code used for the experiments in this manuscript is available as a GitHub repository at https://github.com/fistaco/nascty-cnns (accessed on 3 June 2023).

Acknowledgments: The authors wish to thank Stjepan Picek for the useful comments and discussions regarding both the experiments and the structure of the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

SCA	side-channel analysis
NN	neural network
CNN	convolutional NN
NASCTY-CNN	neuroevolution to attack side-channel traces yielding CNNs
DL-SCA	deep learning-based SCA
GA	genetic algorithm
AES	advanced encryption standard
ASCAD	ANSSI SCA Database
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information
ReLU	rectified linear unit
SELU	scaled exponential linear unit
CCE	categorical cross-entropy
MLP	multi-layer perceptron
NEAT	neuroevolution of augmenting topologies
DENSER	deep evolutionary network structured representation

References

1. Kocher, P.C.; Jaffe, J.; Jun, B. Differential Power Analysis. In Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, Santa Barbara, CA, USA, 15–19 August 1999; Springer: London, UK, 1999; CRYPTO '99. pp. 388–397.
2. Quisquater, J.J.; Samyde, D. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In *Proceedings of the Smart Card Programming and Security*; Attali, I., Jensen, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2001; pp. 200–210.

3. Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*; Springer: Berlin/Heidelberg, Germany, 2006; p. 338. ISBN 0-387-30857-1. Available online: http://www.dpabook.org/ (accessed on 3 June 2023).
4. Benadjila, R.; Prouff, E.; Strullu, R.; Cagli, E.; Dumas, C. Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. *IACR Cryptol. ePrint Arch.* **2018**, *22*, 2018.
5. 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 Trans. Cryptogr. Hardw. Embed. Syst.* **2019**, *2019*, 148–179. [CrossRef]
6. Wu, L.; Picek, S. Remove some noise: On pre-processing of side-channel measurements with autoencoders. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 389–415. [CrossRef]
7. Zaid, G.; Bossuet, L.; Habrard, A.; Venelli, A. Methodology for efficient CNN architectures in profiling attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2019**, *2020*, 1–36. [CrossRef]
8. Wouters, L.; Arribas, V.; Gierlichs, B.; Preneel, B. Revisiting a Methodology for Efficient CNN Architectures in Profiling Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 147–168. [CrossRef]
9. Picek, S.; Perin, G.; Mariot, L.; Wu, L.; Batina, L. SoK: Deep Learning-Based Physical Side-Channel Analysis. *ACM Comput. Surv.* **2023** , *55*, 1–35. [CrossRef]
10. Rijsdijk, J.; Wu, L.; Perin, G.; Picek, S. Reinforcement Learning for Hyperparameter Tuning in Deep Learning-based Side-channel Analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2021**, *2021*, 677–707. [CrossRef]
11. Wu, L.; Perin, G.; Picek, S. I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis. *IACR Cryptol. ePrint Arch.* **2020**, *2020*, 1293. [CrossRef]
12. Standaert, F.X.; Malkin, T.G.; Yung, M. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *Proceedings of the Advances in Cryptology—EUROCRYPT 2009*; Joux, A., Ed.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 443–461.
13. Wu, L.; Won, Y.S.; Jap, D.; Perin, G.; Bhasin, S.; Picek, S. *Explain Some Noise: Ablation Analysis for Deep Learning-Based Physical Side-Channel Analysis*; Cryptology ePrint Archive, Paper 2021/717; 2021. Available online: https://eprint.iacr.org/2021/717 (accessed on 3 June 2023).
14. Eiben, A.E.; Smith, J.E. *Introduction to Evolutionary Computing*, 2nd ed.; Springer Publishing Company, Incorporated: Berlin/Heidelberg, Germany, 2015.
15. Miller, B.L.; Goldberg, D.E. Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise. *Evol. Comput.* **1996**, *4*, 113–131. [CrossRef]
16. Luke, S. *Essentials of Metaheuristics*, 2nd ed.; Lulu: Raleigh, NC, USA, 2013. Available online: http://cs.gmu.edu/~sean/book/metaheuristics/ (accessed on 3 June 2023).
17. Katoch, S.; Chauhan, S.S.; Kumar, V. A review on genetic algorithm: Past, present, and future. *Multim. Tools Appl.* **2021**, *80*, 8091–8126. [CrossRef] [PubMed]
18. Holland, J.H. Genetic algorithms. *Sci. Am.* **1992**, *267*, 66–73. [CrossRef]
19. Deb, K.; Agrawal, S. A niched-penalty approach for constraint handling in genetic algorithms. In *Proceedings of the Artificial Neural Nets and Genetic Algorithms*; Springer: Vienna, Austria, 1999; pp. 235–243.
20. Liagkouras, K.; Metaxiotis, K. An Elitist Polynomial Mutation Operator for Improved Performance of MOEAs in Computer Networks. In Proceedings of the 22nd International Conference on Computer Communication and Networks, ICCCN 2013, Nassau, Bahamas, 30 July–2 August 2013; pp. 1–5.
21. Blank, J.; Deb, K. Parameter Tuning and Control: A Case Study on Differential Evolution With Polynomial Mutation. In Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2022, Padua, Italy, 18–23 July 2022; pp. 1–8.
22. Picek, S.; Heuser, A.; Jovic, A.; Bhasin, S.; Regazzoni, F. The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2019*, 209–237. [CrossRef]
23. Maghrebi, H.; Portigliatti, T.; Prouff, E. Breaking cryptographic implementations using deep learning techniques. In *Proceedings of the International Conference on Security, Privacy, and Applied Cryptography Engineering*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 3–26.
24. TELECOM ParisTech SEN Research Group. *DPA Contest*, 2nd ed.; 2010 .
25. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv* **2014**, arXiv:1409.1556.
26. Real, E.; Moore, S.; Selle, A.; Saxena, S.; Suematsu, Y.L.; Tan, J.; Le, Q.; Kurakin, A. Large-scale evolution of image classifiers. *arXiv* **2017**, arXiv:1703.01041.
27. Stanley, K.O.; Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. Comput.* **2002**, *10*, 99–127. [CrossRef] [PubMed]
28. Assunção, F.; Lourenço, N.; Machado, P.; Ribeiro, B. DENSER: Deep evolutionary network structured representation. *Genet. Program. Evolvable Mach.* **2019**, *20*, 5–35. [CrossRef]
29. Sun, Y.; Xue, B.; Zhang, M.; Yen, G.G. Evolving deep convolutional neural networks for image classification. *IEEE Trans. Evol. Comput.* **2019**, *24*, 394–407. [CrossRef]
30. Knezevic, K.; Fulir, J.; Jakobovic, D.; Picek, S. NeuroSCA: Evolving Activation Functions for Side-channel Analysis. *IACR Cryptol. EPrint Arch.* **2021**, *2021*, 249. [CrossRef]
31. Acharya, R.Y.; Ganji, F.; Forte, D. InfoNEAT: Information Theory-based NeuroEvolution of Augmenting Topologies for Side-channel Analysis. *arXiv* **2021**, arXiv:2105.00117.

32. Schijlen, F. Neuroevolution Applied to Profiled Side-Channel Attacks. Master's Thesis, Delft University of Technology, Delft, The Netherlands, 2022.
33. Deb, K.; Deb, D. Analysing mutation schemes for real-parameter genetic algorithms. *Int. J. Artif. Intell. Soft Comput.* **2014**, *4*, 1–28. [CrossRef]