Mind the Grammar: Side-Channel Analysis driven by Grammatical Evolution

Mattia Napoli[1], Alberto Leporati[2], Stjepan Picek[3], and Luca Mariot[4]

[1]Semantics, Cybersecurity and Services Group, University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands
{m.napoli, l.mariot}@utwente.nl

[2]Department of Informatics, Systems and Communications, University of Milano-Bicocca, Viale Sarca 336/14, 20126 Milano, Italy
alberto.leporati@unimib.it

[3]Digital Security Group, Radboud University, Postbus 9010, 6500 GL Nijmegen, The Netherlands
stjepan.picek@ru.nl

April 17, 2025

Abstract

Deep learning-based side-channel analysis is an extremely powerful option for profiling side-channel attacks. However, to perform well, one needs to select the neural network model and training time hyperparameters carefully. While many works investigated these aspects, random search could still be considered the current state-of-the-art. Unfortunately, random search has drawbacks, since the chances of finding a good architecture significantly drop when considering more complex targets.

In this paper, we propose a novel neural architecture search approach for SCA based on grammatical evolution—SCAGE. We define a custom SCA grammar that allows us to find well-performing and potentially unconventional architectures. We conduct experiments on four datasets, considering both synchronized and desynchronized versions, as well as using feature intervals or raw traces. Our results show SCAGE to perform extremely well in all settings, outperforming random search and related works in most of the considered scenarios.

Keywords Side-channel Analysis, Deep Learning, Evolutionary Algorithms, Neuroevolution, Grammatical Evolution

1 Introduction

Cryptographic algorithms (ciphers) are essential in data communication as they ensure the secrecy of sensitive information. While many ciphers are considered to be theoretically secure, they leak information once they are implemented. Implementation attacks aim to retrieve secret information or bypass security measures by exploiting vulnerabilities specific to how ciphers are implemented in practice. Such attacks are powerful and represent a critical concern in practice [1]. Side-channel Attacks (SCAs) typically represent non-invasive and passive attacks, as they do not inject abnormal signals. More precisely, SCA exploits unintended side-channel leakage occurring during the execution of cryptographic algorithms on targeted devices. Common side-channel information includes power consumption, electromagnetic emissions, and timings [24]. Furthermore, it is common to divide SCA into direct and profiling attacks. Direct SCAs analyze traces from a target device by directly relying on statistical methods. Depending on the attack, direct SCAs can take from a single trace to millions of traces to break a target [28]. On the other hand, a profiling SCA assumes a more powerful attacker with access to an open device similar (ideally, identical) to the target, allowing the attacker to create a profiling model. Profiling attacks are also called two-stage attacks as they unfold in two phases: 1) building a model using the clone device under control and 2) utilizing the model to obtain the secret from the target device. In the last decade, machine (and especially deep) learning emerged as the most powerful option for profiling SCA; see, e.g., [23, 7, 16, 12]. In fact, Deep Learning-based SCA (DL-SCA) showed the capability of breaking protected targets with only a single attack trace, both under masking and hiding countermeasures [26].

For deep learning to be so successful, the first phase of model building must be carried out carefully. This commonly entails at least 1) pre-processing steps such as feature normalization/standardization, feature engineering, and/or data augmentation and 2) model selection, i.e., selecting a neural network (NN) type along with its hyperparameters. This model selection task represents the core challenge for most DL-SCA works [28].[1] While related works explored diverse (and complex) options to address this challenge, including evolutionary algorithms [2], reinforcement learning [29], or Bayesian optimization [38], surprisingly, even a random search can reach an optimal attack performance [26], meaning that it can break a target with a single attack trace.

However, while often successful, random search has also serious drawbacks. First, when using larger and more complex architectures like transformers [12], there are many different potential combinations to choose from, and a random search is unlikely to sample from all relevant parts of the search space since a factor of luck is always involved.[2] Second, as the targets become more difficult

[1] Recently published guidelines for machine learning-based evaluations also underline the importance of hyperparameter tuning [9].

[2] Note that even without sampling all relevant parts of the search space, we could still assume the search to result in a well-performing architecture.

to break, random search struggles to find well-performing architectures [26]. This is intuitive, as higher target difficulty means fewer good architectures capable of breaking the target. Consequently, it can easily happen that only a few architectures out of hundreds of randomly generated ones are actually performing well. Third, a random search still requires precisely defined hyperparameter ranges. Indeed, if the ranges are too small, it could be that there are no well-performing architectures. On the other hand, if the ranges are too large, it may become too difficult for a random search to find a satisfactory architecture. Finally, while a random search can produce any architecture, in practice, this is not the case. A random search is usually constrained by designer-chosen rules that dictate the overall structure of the network, which could, in turn, inhibit the approach from finding novel, less-conventional solutions. Let us take convolutional neural networks (CNN) as an example. Random search commonly works by building a number of convolutional layers that are followed by pooling layers. After those, there are a number of fully connected layers [14]. While this may often work, it prevents the construction of an architecture that starts with, e.g., a pooling layer.

As already mentioned, related works explored diverse ways to build well-performing neural network architectures for DL-SCA. However, there is a trade-off to consider: from one side, there is the expressiveness of the possible solutions and of the reward functions that guide the search process. On the other hand, the computational complexity of the underlying search problem influences the difficulty of finding a good solution.

In this work, we propose a novel approach to address the problem of hyperparameter tuning of deep neural networks for SCA: Side-channel Analysis driven by Grammatical Evolution—SCAGE. This approach, built upon FastDENSER++ [6], is based on Grammatical Evolution (GE), a type of evolutionary algorithm that allows one to evolve computer programs by describing their general structure as a context-free grammar [25]. In SCAGE, the underlying grammar yields a flexible yet expressive medium to define the architecture and hyperparameters of neural network models for DL-SCA. The genotype of a candidate solution in SCAGE specifies a sequence of grammar production rules to construct a complete neural network. An Evolutionary Strategies (ES) approach is then implemented to iteratively tweak a population of neural network genotypes, driving the selection process with an appropriate reward function. The use of ES enables a good balance between exploration (visiting diverse areas of the search space of neural networks) and exploitation (finding well-performing neural network models in specific search space areas). We tested our approach on four datasets, considering masking and desynchronization countermeasures.

The obtained results showcase SCAGE to perform extremely well, outmatching the random search approach and other techniques from related works in most settings. Although the underlying grammar can be easily tailored to a specific attack scenario (e.g., allowing only certain types of layers), by analyzing the architectures of the best networks, we remark that SCAGE is also able to adapt automatically to different scenarios under a generic grammar without enforcing explicit

instructions. In particular, we observe that SCAGE evolves networks without convolutional layers on raw synchronized traces, while it employs them when dealing with the trimmed scenario or when desynchronization is applied.

Contributions. In summary, the main contributions of this paper are as follows.

1. We propose a novel approach to hyperparameter tuning for DL-SCA that leverages grammatical evolution—SCAGE. Our technique is suitable for diverse settings and delivers excellent attack results.
2. Our approach leverages a custom-developed SCA grammar that is robust across all relevant SCA options but also allows for easy adjustments in new scenarios.
3. We thoroughly test SCAGE on four masked datasets with different levels of desynchronization. The results show an excellent attack performance, outmatching current state-of-the-art approaches in most scenarios.
4. We analyze the best architectures evolved by SCAGE, remarking that they often follow an unconventional structure compared to related works.

2 Background

2.1 Deep Learning-based SCA

Side-channel Analysis considers attacks that do not aim at the mathematical weaknesses of a cryptographic algorithm, but rather at those of its implementation [24]. The SCA core idea is to compare some secret data-dependent predictions of the physical leakages and the actual (measured) leakage to identify the data most likely to have been processed. As already stated in Section 1, SCA can be divided into direct attacks and profiling attacks. Deep learning-based SCA in its usual form is an example of a profiling attack. There, one commonly follows the supervised learning paradigm to perform a classification task. The supervised learning paradigm requires that during the training phase, we have labeled examples. The classification task requires the labels to be discrete values. The number of labels is determined by the cipher and the leakage model. For instance, if we consider the AES cipher, due to the divide-and-conquer approach and the fact that AES is a byte-oriented cipher, we can consider every byte value separately. As commonly done, we can use the byte value after the S-box part, which would give us 256 possible values. This scenario is commonly called the Identity leakage model (denoted by ID). On the other hand, if we assume that the implementation leaks in the Hamming weight (or distance) leakage models (denoted by HW/HD), then there are 9 possible values, i.e., one for each Hamming weight that an 8-bit vector can have.

Let us assume we have a training set of inputs x along with their corresponding labels y. Then, our goal is to learn a function f that maps x to the corresponding y. If the function f obtained after the training phase is a good approximation of the relationship between the input and the output, it will generalize to unseen data

when predicting the label y. More precisely, to measure how well the function fits the training data, a loss function L is defined. The loss of predicting value $\hat{y}$ for the (x, y) is $L(y, \hat{y})$. The model is then trained through a learning process to minimize the loss by adjusting its parameters (denoted as θ) using optimization methods [11]. Finally, the model is tested on unseen data (test data) to measure its capability to generalize. While it is common to use metrics such as accuracy or recall in the usual machine learning scenarios, they make less sense in SCA [27]. As such, to assess the attack's effectiveness, we use metrics that directly tell us how many guesses one needs to make before finding the correct key. The simplest metric, key rank, states the position of the correct key in the key guessing vector, i.e., the vector of all possible keys sorted from the most likely to the least likely guess. A more refined metric (and the one we will use throughout the paper) is the guessing entropy (GE) [34], which is the average key rank. The averaging is done to improve the statistical quality (i.e., to reduce the effect of specific traces used) of the attack.

2.2 Neuroevolution

As we mentioned in the previous section, finding the correct set of weights for the connections of a neural network to minimize a specific loss function is addressed through standard optimization algorithms. However, this step still assumes that the overall architecture and the other hyperparameters of the network have been fixed. *Neural Architecture Search* (NAS) is a broad field encompassing various techniques to automate the design of a neural network's architecture with the objective of finding a suitable one to solve a specific task. Elsken et al. [8] observe that a typical NAS problem involves the definition of three components: 1) a suitable search space, 2) the search strategy used to explore the search space, and 3) how to estimate the performance of the architectures explored by the search strategy. In particular, when the search strategy is an evolutionary algorithm, the NAS process is also called *Neuroevolution* [10]. In what follows, we cover the background notions related to evolutionary algorithms and neuroevolution used throughout the paper, focusing in particular on grammatical evolution.

Evolutionary Algorithms and Grammatical Evolution. Evolutionary Algorithms (EAs) is a general term for a class of metaheuristic optimization algorithms loosely inspired by the principles of biological evolution, which tweak a population of candidate solutions for an optimization problem. An EA typically decouples the genotype of the individuals (the representation on which the algorithm operates on) from their phenotype (the final expression of the evolved individuals) and requires a mapping process to perform this translation. In its general form, an EA iteratively selects the best individuals in the population to be reproduced in the next generation. The selection is driven by a fitness function, which evaluates how good a candidate solution is in solving the specific optimization problem. New solutions are then generated and injected in the population by applying crossover and

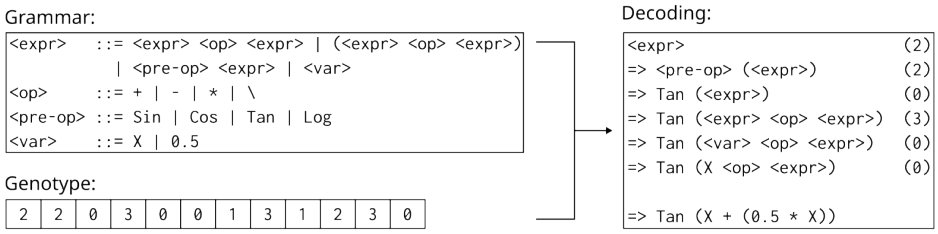


Figure 1: An example of GE decoding, using a grammar for evolving expressions.

mutation operators on the selected parents. The most famous type of EAs are perhaps Genetic Algorithms (GAs) [13], where the genotype of the solutions is usually represented through fixed-length bitstrings. In Genetic Programming (GP) [18], instead, the aim is to directly evolve computer programs encoded by syntactic trees whose leaves represent the inputs to the program, while the internal nodes are operators combining the values of these inputs. The output of the program is then evaluated at the root node.

Grammatical Evolution (GE) was first introduced by O'Neil et al. [25] as a variant of GP, which leverages context-free grammars to represent the candidate programs instead of syntactic trees. At the core of GE, there is a formal language expressed using a grammar in the Backus-Naur Form (BNF) notation. In particular, every production rule of the grammar is written in the form `<non-terminal> ::= <expression>`. The symbol on the left side of `::=` is called a *non-terminal*, and, during the derivation process, it is replaced by one of the options on the right. The right-hand side of the rule can include other non-terminal symbols (which are recursively substituted), terminals (i.e., actual tokens of the language), and the `|` symbol that separates alternative substitutions. Each rule has a number of possible choices, and GE employs a Genetic Algorithm to select which rule to use at every junction in the derivation process. In GE, every individual is represented as a variable-length list of randomly generated numbers. When mapping the genotype to the phenotype, every time a decision has to be made during the derivation process, the next number is read from the chromosome, and that determines the rule to apply. Figure 1 provides an example of such a mapping process.

DENSER Approach to Neuroevolution. While there are several strategies to search for the optimal architecture of a neural network by means of EA (e.g., the NeuroEvolution Augmenting Topologies approach, or NEAT [35]), here we focus only on those techniques based on grammatical evolution.

The initial work that leveraged grammars and evolutionary algorithms to search for Convolutional Neural Networks (CNNs) was DENSER [4]. The genotype of an individual in DENSER is an ordered linear structure where each position is a functional unit (called *module*) of the corresponding network, and it is mapped to a symbol in the grammar. By choosing a grammar-based approach, these units can

```
<features> ::= <convolution> | <pooling>
<convolution> ::= layer:conv1d [num-filters,int,1,4,512]
<pooling> ::= <pool-type> [kernel-size,int,1,2,20] [stride,int,1,2,20]
<pool-type> ::= layer:pool-avg1d | layer:pool-max1d
...
<classification> ::= <fully-connected> | <dropout>
...
<learning> ::= <rmsprop> <early-stop> [batch_size,int,1,50,500]
             | ...
...
```

Listing 1: Part of the grammar for evolving CNNs in FastDENSER++.

encode any component of the network that can be expressed using derivation rules: in DENSER, this feature is also used to evolve hyperparameters like the optimization algorithm to minimize the loss, the learning rate, and the batch size. Every unit in the genotype is encoded similarly to Dynamic Structured Grammatical Evolution (DSGE) [3], which means that it encodes the expansion possibilities of the grammar for that unit.

Basically, an individual has two genotypic levels: an outer one, such as `<feature> <feature> <classification> <learning>`, which encodes the architecture of the network and defines the starting symbol for every module; and an inner one, which encodes the expansion possibilities and the numerical values required by the single module. To ground our discussion with a specific example, let us consider the grammar from Listing 1 and the following (partial) phenotype, `layer:pool-max1 kernel-size:4 stride:2 <feature> <classification> <learning>`. The inner genotype for the first `<feature>` module contains the indices to pick (in order) the second expansion possibility (for the pooling layer), then the first one (and only one for `<pooling>`), and finally the second one (to get `layer:pool-max1`). However, the `<pooling>` symbol also requires numerical values for `kernel-size` and `stride`, and therefore, the genotype also contains values for those parameters: in this specific example, the values are 4 and 2, respectively.

FastDENSER [5] is the successor of DENSER, and it was proposed as a faster alternative with a few major changes; one of them is a new graph-like representation that enables the evolution of skip connections. This is achieved by adding another level in the genotype to encode the connections, i.e., to which layers' outputs the current layer has access to. A second change, which is the one responsible for the speed-up, is the introduction of the $(1+\lambda)$ Evolution Strategy (ES) [22] instead of a classic GA. This means that FastDENSER has a population of $1+\lambda$ individuals, and only one is selected from the population to reproduce (i.e., the one with the best fitness). Then, λ children are generated by randomly mutating the selected parent. Finally, these λ children join the parent to form the next generation.

Finally, FastDENSER++ [5] is a direct extension of FastDENSER. The core idea of this new version is to set the maximum training time independently for each individual. Indeed, in FastDENSER, it could be the case that not enough time is granted for an NN to fit the dataset properly, although the evolved topology and

hyperparameters are promising. By allowing the training time of single individuals to grow as needed, FastDENSER++ can evolve fully trained neural networks that are ready for deployment.

2.3 Datasets

Below, we briefly describe the four datasets used to evaluate our approach for DL-SCA. In particular, we selected these datasets for the sake of comparison with state-of-the-art approaches recently published in the literature [26, 12, 15].

ASCADf. The ASCADf dataset[3] contains measurements from an 8-bit AVR microcontroller, where the leakage sources are the electromagnetic emissions from the chip. The AES-128 implementation is protected with first-order Boolean masking, except for the first two bytes that are left unprotected for testing purposes. The acquisition window covers only the first round of AES and each trace consists of 100 000 sample points. The dataset contains 60 000 traces overall, which are split into two groups: 50 000 for the profiling phase and 10 000 for the attack. All encryption operations in both the profiling and attack groups are performed using the same *fixed* key, hence the name of the dataset.

ASCADr. The ASCADr dataset[4] has a similar setup to ASCADf concerning the measurements source. The main difference stems from the fact that the traces in the profiling group of ASCADr use *random* keys to perform the encryption instead of a fixed one. The traces are also bigger as they consist of 250 000 samples. In total, 200 000 traces are provided for the profiling phase and 100 000 for the attack phase. The encryption for the attack group is still performed using a fixed key.

CHES CTF 2018. The CHES CTF 2018[5] dataset contains power measurements from ARM Cortex-M4 devices running a masked AES implementation. Each trace consists of 650 000 samples, but the analysis is commonly limited to the first 150 000, which includes the first round of AES encryption. The traces are provided in four sets of 10 000 traces each: the first three use random keys and therefore are combined to form the profiling set, while the remaining one uses a fixed key and is left as the attacking set.

eShard. The last dataset, eShard[6], also features an AES implementation protected with first-order Boolean masking running on ARM Cortex-M4 devices. The

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

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

[5] https://zenodo.org/record/3733418#.Yc2iq1ko9Pa

[6] https://gitlab.com/eshard/nucleo_sw_aes_masked_shuffled

raw traces are not publicly available, and the authors only provide a trimmed version of 1 400 sample points. The dataset contains 100 000 traces in total, without any distinction on profiling and attack traces. Every encryption operation is performed using the same fixed key.

3 Related Work

From the first work on deep learning-based SCA [23], the SCA community invested significant effort into deploying more powerful and automated attacks. In fact, a careful review indicates that finding effective neural network architectures still represents the largest part of the published works [28]. Moreover, while in DL-SCA, one can use diverse neural network architectures, common choices today are the multilayer perceptron (MLP) and the convolutional neural network (CNN).

To design such architectures, we can recognize several characteristic directions. First, works like [42] and [37] aim to find methodologies for designing neural network architectures for SCA. The main advantage of such approaches is that, if successfully designed and applied, they can significantly simplify the process of finding good neural network architectures. At the same time, the main drawback seems to be the relative fragility of such approaches and the difficulty in adapting them to different threat models and targets. Next, it is possible to use diverse search strategies to design well-performing neural network architectures. Among the various approaches is Random Search (RS), where one randomly constructs a large number of architectures and uses the best ones. While simple, RS shows good performance, and in some cases, even state-of-the-art results are achieved, see, e.g., [26]. That work shows that when RS is applied to *raw* traces, it discovers models for the ASCAD dataset able to recover one byte of the key with a single attack trace, making it an optimal attack. Naturally, there are also more advanced strategies, like reinforcement learning [29], Bayesian optimization [38], or evolutionary algorithms [23]. The main advantage of such approaches is that they can often find better-performing architectures (compared to random search and in cases where random search does not perform as well) but at the cost of a more complex setup and significantly higher computational requirements.

The success of deep learning-based attacks does not depend exclusively on the underlying architecture but also on factors like data augmentation [20] or feature selection. There, common options include working with an interval of traces [7] or raw traces [21]. Interestingly, even in the case of raw traces (it is intuitively clear that they can offer more information, but the process becomes computationally more complex), state-of-the-art results are still achieved with a random search [26], and the final architectures are relatively small. However, as the results from [26] report, the main problem of a random search is the low success rate. In the first place, this means that it is not easy to randomly generate a model capable of recovering the key and, more importantly, it does not imply that such a model can perform this task efficiently, i.e., within a low number of attack traces. These short-

comings become more evident when countermeasures such as desynchronization are introduced: in such cases, the success rate in [26] does not go above 3.05% for the ASCADf, ASCADr, and CHES CTF 2018 datasets.

Other works that achieve comparable performance without resorting to random search often use more complex models: [12] and [21] chose transformer networks to perform the attacks, [15] used Conditional Generative Adversarial Networks (CGANs) to leverage knowledge from previous datasets, and [19] successfully combined language models with multitask learning to attack ASCAD. It is worth mentioning that instead of concentrating on architectures (of course, their tuning is still an important factor), it is possible to consider different threat models, like weakly profiling attack [39], leakage model-flexible attack [40], non-profiling attack [36], and collision attack [33, 41].

Finally, looking into evolutionary algorithm-based approaches, and more specifically the branch of neuroevolution, there are a number of works that showcase the potential of such an approach. Even the first work on deep learning-based SCA [23] reported the usage of genetic algorithms for hyperparameter tuning (however, without providing details about the settings). Knezevic et al. used neuroevolution to evolve custom activation functions for SCA [17]. On the other hand, Rioja et al. used estimation of distribution algorithms to select points of interest [30].

InfoNEAT [2] is currently the best-performing neuroevolution framework for SCA. Its main feature is the ability to discover so-called augmenting topologies: as the algorithm operates at the level of individual neuron connections and their weights, it can find irregular configurations that a human could hardly think of. Moreover, it employs a One-vs-All approach for multi-class classification by training 256 submodels, one for each value of the target sub-key. The output of the submodels is then combined into another classifier to improve the predictions.

An alternative to InfoNEAT is NASCTY [31], which is based on genetic algorithms. NASCTY operates at a higher abstraction level: it uses a list of blocks as the genome, which encodes the hyperparameters for convolutional and dense layers. However, it does not perform as well as InfoNEAT, and it enforces a rigid structure of the network's topology, which can prevent the discovery of alternative combinations that may outperform conventional ones.

4 SCAGE: SCA driven by Grammatical Evolution

As our objective is to implement a neuroevolution framework that could replace random search to efficiently generate SCA models, neither InfoNEAT nor NASCTY seems to be a suitable candidate to build upon. Indeed, the former focuses on individual connections and requires many submodels, while the latter imposes too many constraints on the networks' architecture. Hence, we decided to adapt FastDENSER++ [6] as a flexible and configurable neuroevolution approach to evolve good architectures for DL-SCA. FastDENSER++ resembles NASCTY in that it operates at a higher layer of abstraction by focusing on the layers as the units of

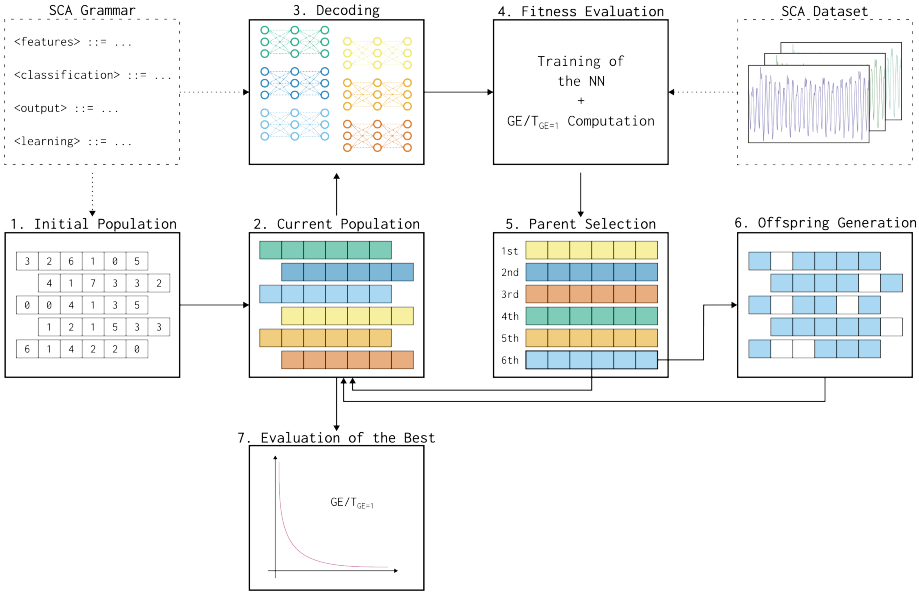


Figure 2: The evolutionary approach of SCAGE.

evolution. In fact, the focus of FastDENSER++ is to evolve both the architecture of the neural networks and the hyperparameters of its individual layers.

FastDENSER++ is the latest version of a NAS tool specifically tailored to evolve neural networks for image classification tasks. In this section, we present SCAGE—Side-channel Analysis driven by Grammatical Evolution—which is our variant of FastDENSER++ remodeled to suit the generation of neural network models for SCA. Figure 2 provides an overview of the evolutionary approach implemented by SCAGE. In the remainder of this section, we describe each step of the approach in detail.

SCA Grammar. The first significant step to tackle is the definition of a new grammar to suit the generation of SCA models, specifically CNNs. The grammar is inspired by Perin et al.'s work [26], with the notable difference that the parameter ranges are broadened, and more flexibility is given to the structure of the networks. We made this choice to provide the framework with a larger search space, encompassing both state-of-the-art networks discovered by random search as well as unconventional topologies and hyperparameter combinations. The resulting grammar is shown in Listing 2. One of the main features of the grammar is that it does not enforce any specific order of the layers: each `<features>` or `<classification>` symbol is derived independently from the others. Therefore, the choice of the previous layer type does not have any effect on the rest of the network. Notice that, in principle, we could impose certain patterns; for instance, by changing the rule of `<features>` in `<features> ::= <convolution>`

```
<features> ::= <convolution>
             | <pooling>
             | <batch-norm>
<convolution> ::= layer:conv1d [num-filters,int,1,4,512]
    ↪ [filter-shape,int,1,2,80] [stride,int,1,30,50]
    ↪ <activation-function>
<pooling> ::= <pool-type> [kernel-size,int,1,2,20] [stride,int,1,2,20]
<pool-type> ::= layer:pool-avg1d | layer:pool-max1d
<batch-norm> ::= layer:batch-norm
<classification> ::= <fully-connected> | <dropout>
<fully-connected> ::= layer:fc <activation-function> <regularizer>
    ↪ [num-units,int,1,10,1000]
<regularizer> ::= <regularizer-type> [regrate,float,1,0.00001,0.05]
<regularizer-type> ::= reg:l1 | reg:l2 | reg:none
<dropout> ::= layer:dropout [rate,float,1,0.05,0.5]
<activation-function> ::= act:relu | act:selu
<output> ::= layer:fc act:softmax num-units:256 reg:none
<learning> ::= <optimizer> [lr,float,1,0.0001,0.001] <early-stop>
    ↪ [batch_size,int,1,100,1000] epochs:100
<optimizer> ::= learning:adam | learning:rmsprop
<early-stop> ::= [early_stop,int,1,5,20]
```

Listing 2: Grammar for evolving CNNs in SCAGE. Non-terminals in bold are starting symbols, i.e., those that start the derivation process, and they correspond to the *modules* in the genotype. The conditions for the numerical values required by certain hyperparameters are specified within square brackets: for example, `[num-filters, int, 1, 4, 512]` means that 1 integer value between 4 and 512 is generated for the number of filters.

`| <convolution> <pooling> | <convolution> <pooling> <batch-norm>` we would obtain an architecture similar to the VGG convolutional neural network [32], and also more similar to those explored until now. However, we decided to adopt a more general grammar, allowing SCAGE to experiment also with unconventional combinations that do not necessarily follow known architectures.

Initial Population. We initialize all networks by choosing uniformly at random the derivation rules to apply and the numerical values for the remaining hyperparameters within the boundaries defined in the grammar. This process is repeated for λ times: the initial population is, in fact, the only one that does not have size of $1+\lambda$ since the parent has not been selected yet.

The hyperparameters of the evolution process also include a minimum and maximum number of layers per module type during the initialization. In the original version of FastDENSER++, these values are 2, 3, or 4 for the `features` modules and 1 for the `classification` modules. However, during preliminary tuning experiments we found that initializing the SCA models with 3, 4, or 5 `features` modules and 1, 2, or 3 `classification` modules allowed SCAGE to generate a better starting population, which in turns allows the evolution process to converge faster. The initial maximum training time for a network has also been reduced from 600 seconds of FastDENSER++ to 400 seconds. The reason for this choice

is twofold: first, preliminary experiments indicated that 400 seconds was enough for the majority of the models to complete their training; second, we decided to let SCAGE figure out whether a network would require more training time by following its evolutionary process.

Decoding Procedure. The decoding step works as in FastDENSER++, with the main difference being the removal of the logic to encode skip connections. The motivation for this change is that most of the works in the field of DL-SCA [42, 38, 26] do not use this kind of connection. In particular, we focus on discovering unconventional combinations of layers and hyperparameters for CNNs while maintaining a linear topology for the neural networks. The phenotype generated from the genome is a string description of the network that is later parsed during the evaluation phase. The grammar provides the rule for this mini-language that encodes the structure of the NNs. An example of phenotype is the following:

```
layer:conv1d num-filters:469 filter-shape:68 stride:35 act:selu
layer:pool-avg1d kernel-size:14 stride:13 layer:batch-norm layer:fc act:relu
reg:l1 num-units:154 layer:dropout rate:0.3318447376367588 layer:fc
act:softmax num-units:256 learning:rmsprop lr:0.0001 early_stop:19
batch_size:977 epochs:100
```

Fitness Evaluation. Before proceeding with the evaluation, the neural networks have to be trained. The evolution process of FastDENSER++, however, requires four sets in total: a *training* set, a *validation* set to track the overfitting of the network during training and eventually trigger an early stopping, a *test* set to assess the performance of the model after training, and a *final test* set that is used for the best network discovered during the evolution to produce the final metrics. Therefore, for SCA datasets that distinguish between profiling and attack traces, the strategy to generate the four splits is the following: *profiling* traces are divided into training and validation sets; *attack* traces are divided into test and final test sets.

FastDENSER++ originally uses the accuracy of the resulting networks as a fitness score to rank the candidates in the population. However, as mentioned in Section 2.1, accuracy is not a suitable metric for the SCA setting. Hence, in SCAGE, we decided to adopt the guessing entropy (GE) as a fitness function. To compute the GE, we take the average of 100 key ranks computed over 3 000 traces randomly selected from the test set. Remark that, during the computation of the fitness, multiple models may achieve $GE = 1$, thus recovering the key. In case such ties occur, we use the $T_{GE=1}$ metric, i.e., the number of traces required to get $GE = 1$.

In theory, one could directly use the $T_{GE=1}$ metric as a fitness function. However, this strategy would lead to a significant problem: whenever the networks under comparison fail to recover the correct key, even after processing the whole test set, they would all report the same (maximum) value for the $T_{GE=1}$ metric, namely the size of the test set. To resolve this issue, the fitness function of SCAGE is essentially split into three stages:

- First, we evaluate the GE on the whole test set: the best network is the one achieving the lowest GE after processing all traces; if a network manages to recover the key, then it has $GE = 1$, and the evaluation would need to consider the second stage.
- Only if the network achieves $GE = 1$ then we consider $T_{GE=1}$: the best network here is the one that uses the least number of traces.
- Finally, if multiple networks recover the key with the same number of traces, we use the GE score computed just before it gets to 1 as a tie-breaker. For example, if two NNs achieve GE of 1 with 20 traces, we compare their GE using 19 traces. The rationale here is that the lower this value, the closer the network was to recovering the key; therefore, it should be preferred as a candidate to evolve.

Parent Selection. Since we use an evolutionary strategy of the form $(1+\lambda)$-ES, we only need to choose one parent for the next generation. The strategy is simple: we pick the individual that has the lowest fitness, since in the SCA setting the optimization objective is to minimize GE and $T_{GE=1}$. This can potentially lead to re-selecting the same individual over and over, until one of the children can achieve a better fitness. This form of elitism helps preserve the best model that has been discovered so far. Without it, given the limited number of individuals in the population, it could easily happen that none of the children perform better than the parent.

Offspring Generation The offspring is generated by applying only mutation operators to the parent selected in the previous step. Each operator is not applied deterministically but rather stochastically with a specific mutation probability. The number of generated children is λ. While in FastDENSER++ λ is equal to 4, here for SCAGE we set $\lambda = 5$, to favor the exploration of more SCA models from the same parent. Compared to the original FastDENSER++, we kept all mutation operators except those related to skip-connections (as we removed those from the decoding procedure) and the operator that re-uses the same genome of an existing layer when adding a new one to the network. We observed in preliminary experiments that this operator was not providing any benefit in the evolution process. In summary, these are the mutation operators in SCAGE along with the respective probabilities P of applying them (bold values indicate a difference from FastDENSER++):

- *add layer* ($P = 0.25$): adds a new layer to one of the modules;
- *remove layer* ($P = 0.25$): removes a layer from one of the modules;
- *DSGE mutation* ($\mathbf{P = 0.3}$): randomly change one of the parameters of a module. The probability was increased from 0.15 to 0.3 to favor the exploration of NNs with the same layers but with different hyperparameters;
- *macro mutation* ($\mathbf{P = 0.2}$): like the DSGE mutation but applied on "macro" modules, such as `<learning>`; this was reduced from 0.3 to 0.2 as prelimi-

nary experiments suggested that the final performance was not so dependent on the configuration of these learning hyperparameters;

- *increase time* ($P = 0.2$): does not mutate the network but trains it for longer.

Except for the *increase time* operator, every other mutation can occur independently of the others, potentially leading to children with multiple changes to the genotype of the parent. During the evolution, the mutation operators that change the number of modules are applied only if the final genome respects the boundaries on the number of layers. The remaining operators act according to the definitions provided in the grammar.

Stopping Criteria and Best Model Evaluation. A perfect model can recover the correct key using only one attack trace. Therefore, the computation is halted when one of the evolved networks achieves such a fitness score. Otherwise, the evolution continues until a maximum number of generations is reached. We set this value to 30 generations, reducing the original value of 150 in FastDENSER++, as preliminary experiments suggested that it was a good trade-off between the possibility of finding a successful model and the time required to run the framework. After the stopping criteria are met, the best model found during the evolution is tested again on a different set of attack traces, the *final test* set, which is not used during the evolution process.

5 Experimental Results

5.1 Experimental Setup

Every experiment was run in a high-performance computing (HPC) cluster with access to 2 CPU cores, one NVIDIA L40 GPU, and 90+ GB of RAM. For our implementation, we relied on Python 3.10 and Tensorflow 2.14.1. We tested SCAGE over the four datasets described in Section 2.3, namely ASCADf, ASCADr, CHES CTF 2018, and eShard. The scenarios we considered for the traces are the following:

- *trimmed*: available for ASCAD and eShard, this is a reduced version containing the points of interest with the highest SNR;
- *raw*: available for ASCAD and CHES CTF 2018, but not for eShard; to reduce the dimensionality they are resampled as in [26];
- *desynchronized raw*: with δ_{max} up to 50, 100, and 200; these desynchronization levels were applied to the raw traces before the resampling process.

For each combination of dataset and traces scenario, we repeated the search with SCAGE for 30 independent runs to get statistically sound results.

Another important step when working with SCA measurements is the preprocessing of the traces, as demonstrated by Wouter et al. [37], who tested different approaches. In our implementation, we followed the standardization preprocessing approach, which appeared to be slightly more consistent in the experiments

Dataset	$T_{GE=1}$				
	SCAGE	RS	[26]	[12]	[2]
trimmed	70	115	87	–	130
raw $\delta_{max} = 0$	1	118	1	13	–
raw $\delta_{max} = 50$	3	x	–	–	–
raw $\delta_{max} = 100$	4	x	36	12	–
raw $\delta_{max} = 200$	10	x	–	9	–

Table 1: $T_{GE=1}$ of the best models found by SCAGE on ASCADf compared to other approaches.

discussed in [37]. It is feature-based and instance-based on, respectively, synchronized and desynchronized traces. Finally, all results in the sections are produced using the Identity leakage model. This is in line with recent works [42, 37, 2, 12, 15] that chose this leakage model for their analysis.

As a baseline for comparison, we employed Random Search (RS). Specifically, for each considered combination of dataset and trace scenario, we randomly generate 500 models, as done in [26]. On the other hand, a single run of SCAGE evaluates 150 models, since as described in Section 4 an offspring of $\lambda = 5$ individuals is produced in each generation, and the algorithm runs for 30 generations. For the comparisons with other works, we used the results reported in the corresponding papers.

5.2 ASCADf

In Table 1, we compared the $T_{GE=1}$ values of the best model found by SCAGE over 30 runs and by random search over 500 attempts. We also include the results from the work on feature selection by Perin et al. [26], which is the first one that broke ASCAD with a single trace. Further, we include the results obtained by the transformer network Estranet [12], which provides a wide analysis of different desynchronization levels with competitive results, and from InfoNEAT [2], which is the best performing neuroevolution approach for DL-SCA published so far. Note that EstraNet is not trained on *raw* traces but rather on a window of 10 000 sample points, which enlarges the *trimmed* dataset. We still decided to provide the comparison on the *raw* scenario, as the authors of [12] claim that the attack could be executed on the full trace by repeating the attack over different segments. In this table and in the following ones, the x mark indicates that the model for that scenario is not able to recover the key, while the – symbol means that results are not available for that combination of related work and trace setup.

SCAGE finds the best model for every scenario, except for *raw* traces with $\delta_{max} = 200$, where it requires just one trace more than EstraNET to achieve a guessing entropy of 1. On the other side, a random search fails to discover an appropriate model when applied to desynchronized traces. On aligned traces, RS finds models

	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}
$T_{GE=1}$ of the best model found by SCAGE for every k_i	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Search Success Rate	70%	73%	70%	100%	70%	63%	53%	76%	66%	53%	63%	63%	66%	73%
$T_{GE=1}$ of SCAGE's best model for k_2 retrained for every k_i	1	2	2	1	1	2	x	1	9	1	818	153	7	3
$T_{GE=1}$ of [26]'s best model for k_2 retrained for every k_i	1	x	2	x	x	x	x	x	1	1	x	x	3	x

Table 2: Comparison on all key bytes for ASCADf.

that can recover the key, but their $T_{GE=1}$ is far from the results of SCAGE.

The target of attacks for the ASCAD datasets is usually the output of the third S-box during the first round of AES, which allows to recover the third byte of the key. We verified if SCAGE's behavior is consistent on all the other bytes of the key, except the first two since they are not protected. Table 2 reports the $T_{GE=1}$ of the best model found for every byte and the corresponding search success rate, i.e., the percentage of successful runs of SCAGE. One can see that SCAGE always manages to evolve a model that recovers the corresponding byte of the key using only a single trace. To compare how flexible the models found by SCAGE are, we also retrained the best one evolved for the third byte on all other bytes, as done in [26]. This model can successfully recover the correct value for every byte except one, and for 11 of them, it does so in less than 10 traces. This finding suggests that the best network architectures found by SCAGE are versatile enough to adapt to different bytes. Combining these results with the performance obtained by SCAGE when run on a specific byte, we argue that SCAGE could provide an efficient search approach to recover the full key: the strategy would be to start by targeting an individual byte with SCAGE, retrain the best evolved model for all the others, and then only repeat the search with SCAGE when such a model fails to meet the desired $T_{GE=1}$.

Table 3 provides a more in-depth comparison of the average performance of SCAGE against a random search. It can be observed that SCAGE achieves on average $T_{GE=1}$ scores that are much lower than a random search. More importantly, the search success rate of SCAGE improves over RS in all scenarios, suggesting that evolutionary strategies can investigate the search space of SCA models more efficiently.

Dataset	Average $T_{GE=1}$		Search Success Rate	
	SCAGE	RS	SCAGE	RS
trimmed	115	792	100%	20%
raw $\delta_{max} = 0$	25	648.33	70%	1.2%
raw $\delta_{max} = 50$	168	x	16.6%	0.0%
raw $\delta_{max} = 100$	4	x	3.3%	0.0%
raw $\delta_{max} = 200$	58	x	6.6%	0.0%

Table 3: Average $T_{GE=1}$ of SCAGE versus random search on ASCADf.

Dataset	$T_{GE=1}$					
	SCAGE	RS	[26]	[12]	[2]	[20]
trimmed	23	85	78	–	120	–
raw $\delta_{max} = 0$	1	1	1	5	–	–
raw $\delta_{max} = 50$	1	2	–	–	–	33
raw $\delta_{max} = 100$	1	x	73	5	–	251
raw $\delta_{max} = 200$	1	x	–	4	–	44

Table 4: $T_{GE=1}$ of the best models found by SCAGE on ASCADr compared to other approaches.

5.3 ASCADr

The analysis on ASCADr has the same setup as ASCADf. Table 4 collects the $T_{GE=1}$ of the best models found by SCAGE over 30 runs and by RS. This time, we also included the work on data augmentation from Li et al. [20], as they tested their approach on ASCADr, including different levels of desynchronization. SCAGE finds the best model for the *trimmed* scenario by recovering the key in only 23 traces. Further, SCAGE consistently manages to break the desynchronization countermeasure with δ_{max} up to 200 by requiring one single trace for every considered level. Compared to ASCADf, in this case, the random search can find models when $\delta_{max} = 50$ but fails with higher levels of desynchronization.

Another remark comes from the performance of SCAGE compared to the models discovered in [20], which are still found using a random search. Despite not using data augmentation to train the networks, our evolutionary approach can consistently produce models with state-of-the-art values for $T_{GE=1}$. Therefore, it seems that data augmentation can help when working with sub-optimal models, but it cannot replace an efficient architecture. This is also in line with the ablation study performed by the authors of EstraNet [12]. Indeed, when their transformer is trained without data augmentation, its $T_{GE=1}$ scores are barely affected.

Table 5 reports the results of running SCAGE on all bytes of the key for ASCADr, except the first two, as they are unprotected. As for ASCADf, SCAGE can find models with $T_{GE=1} = 1$ for every key byte. For ASCADr as well, we retrained the best model found by targeting the third byte on all the others and compared

	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}
$T_{GE=1}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Search Success Rate	90%	68%	83%	96%	70%	67%	66%	66%	80%	75%	100%	100%	80%	75%
$T_{GE=1}$ of SCAGE's best model for k_2 retrained for every k_i	2	3	1	1	1	2	2	1	x	1	x	2	1	2
$T_{GE=1}$ of [26]'s best model for k_2 retrained for every k_i	2	10	1	1	5	7	4	1	3	1	19	7	1	3

Table 5: Comparison on all key bytes for ASCADr.

Dataset	Average $T_{GE=1}$		Search Success Rate	
	SCAGE	RS	SCAGE	RS
trimmed	85	771	100%	17%
raw $\delta_{max} = 0$	168	365	90%	0.6%
raw $\delta_{max} = 50$	147	961	40%	0.6%
raw $\delta_{max} = 100$	15	x	26.6%	0.0%
raw $\delta_{max} = 200$	77	x	15.3%	0.0%

Table 6: SCAGE versus random search on ASCADr.

the results with the same setup from [26]. This time, the retraining works for 12 bytes, where the model manages to recover the corresponding key byte in at most three traces. On the other hand, the model found by [26] works on every byte of the key: this simply implies that this specific model discovered by SCAGE is not suitable for all k_i. In such situations, we can re-run an explicit search with SCAGE targeting the required byte.

Finally, Table 6 provides the detailed comparison with RS on average $T_{GE=1}$ and search success rate. SCAGE consistently outperforms RS and achieves better success rates even for high levels of desynchronization. We also note a remarkable improvement when compared to the same results from ASCADf. The combination of a much larger dataset (200 000 traces vs 60 000) and the use of random keys in the profiling stage are probably the main reasons for the improved performance.

5.4 CHES CTF 2018

Since with the CHES CTF 2018 dataset, we are only provided with *raw* traces, we focused our analysis on those and tested different levels of desynchronization. Initially, we attempted to run SCAGE on synchronized traces in the same configuration as on ASCAD, without success. This behavior aligns with the results from [26]: their random search could not find a successful CNN when using the Identity leakage model on *raw* traces. Nevertheless, when they search for

MLPs in the same scenario, they had much better results, suggesting that MLPs are better suited for this combination of dataset and trace type. Therefore, we reconfigured SCAGE to avoid any convolutional layer in the genotype: in practice, we removed the `<feature>` grammar symbol from the setup. We also increased the maximum number of `<classification>` layers to compensate for the missing convolutional ones. Finally, we reduced the maximum number of neurons per dense layer from 1 000 to 400 (as in [26]) to avoid the generation of huge MLPs. These changes alone did not help SCAGE in discovering effective models against the CHES CTF 2018 dataset. A more in-depth analysis revealed that many neural networks were training only for a few epochs due to the intervention of the early-stopping mechanism. While it worked for ASCAD, it seems that CHES CTF 2018 affects the performance of SCAGE, preventing the neural networks from learning the dataset. Hence, we adjusted the grammar by modifying the derivation rules for the `<learning>` symbol and by removing the option for early-stopping. This finally allowed SCAGE to discover successful MLPs on *raw* traces. However, when we test for different desynchronization levels, we use CNN models, as convolutional layers seem necessary to defeat this countermeasure [42].

The case of the CHES CTF 2018 dataset highlights one of the main properties of SCAGE: its flexibility in the configuration and the rich expressivity provided by the grammar. Our work focuses on CNNs by choice, but it can be immediately adapted to other types of neural networks, like MLPs, and potentially extended to even more diverse architectures. The role of the grammar in this context is to provide a clear and effective mechanism to define the search space to suit our needs, and the changes we reported are an example of how easily this can be achieved.

Table 7 reports our results on CHES CTF 2018. On *raw* traces, SCAGE is on par with the best MLP found by [26] and [15]. We can also see a reduction in the performance as the level of desynchronization rises, up to the point where, for $\delta_{max} = 200$, SCAGE cannot find a successful model. Similarly to ASCAD, the models found by a random search on aligned traces have high values of $T_{GE=1}$. On desynchronized traces, RS could not find any model that breaks the target. Overall, it seems that the CHES CTF 2018 dataset is harder to attack compared to ASCAD. This is also confirmed by the average $T_{GE=1}$ and search success rates collected in Table 8. SCAGE outperforms random search by quite some margin, but it is not able to achieve the same success rate obtained on both ASCADf and ASCADr.

5.5 eShard

Table 9 reports the result of running SCAGE against every byte of the key in eShard. To the best of our knowledge, we are the first to successfully retrieve the key using the Identity leakage model on this dataset. The $T_{GE=1}$ score of the models evolved by SCAGE is mainly between 100 and 200 traces (with a few exceptions), and the search success rate varies according to the target. We also tried to retrain the model found by SCAGE on k_0 and found that it works on 15 out of 16 of the bytes, with slightly worse $T_{GE=1}$ scores. We also reported $T_{GE=1}$ scores from [15],

Dataset	$T_{GE=1}$			
	SCAGE	RS	[26]	[15]
raw $\delta_{max} = 0$	28	1 606	13	22
raw $\delta_{max} = 50$	123	x	–	–
raw $\delta_{max} = 100$	715	x	906	–
raw $\delta_{max} = 200$	x	x	–	–

Table 7: $T_{GE=1}$ of the best models found by SCAGE on CHES CTF 2018 compared to other approaches.

Dataset	Average $T_{GE=1}$		Search Success Rate	
	SCAGE	RS	SCAGE	RS
raw $\delta_{max} = 0$	313	1 606	56.6%	0.2%
raw $\delta_{max} = 50$	565	x	10.0%	0.0%
raw $\delta_{max} = 100$	715	x	3.3%	0.0%
raw $\delta_{max} = 200$	x	x	0.0%	0.0%

Table 8: SCAGE versus random search on CHES CTF 2018.

which the authors obtained using their CGAN architecture under the Hamming Weight leakage model. The main difference is that SCAGE is more consistent on different bytes and generally performs better.

We also compared SCAGE to RS on the first byte, k_0. While the absolute performance is comparable, SCAGE's best $T_{GE=1}$ is 136, and RS's best is 179. Moreover, random search has a success rate of 3.80%, while SCAGE reaches 30%. Following the trend of ASCAD and CHES CTF 2018, SCAGE can find successful models in a reliable manner on eShard, compared to other techniques.

6 Discussion

The results presented in the previous section show that the models discovered using SCAGE are overall better than those of the state-of-the-art models published in the literature for all the considered datasets. More importantly, they highlight the advantages of using an evolutionary approach in designing the architecture of a neural network compared to a random search.

Search success rate. The main benefit of SCAGE is its search success rate, which is consistently better than a random search in all considered scenarios. We argue that this performance results from the iterative tweaking strategy inherent to evolutionary algorithms: by slightly mutating the best networks, SCAGE has higher chances of preserving their useful parts while modifying those that are sub-optimal. On the other hand, a random search generates a new model from scratch at each step, forgetting what it found before.

	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}
$T_{GE=1}$ of the best model found by SCAGE for every k_i	136	134	153	162	220	172	152	142	232	135	135	167	143	133	118	163
Search Success Rate	30%	40%	16%	23%	13%	23%	26%	30%	16%	33%	40%	30%	26%	26%	36%	23%
$T_{GE=1}$ of SCAGE's best model for k_0 retrained for every k_i	178	195	x	188	245	177	183	163	334	222	142	168	197	220	144	177
[15] CGAN-SCA (ref: ASCADr)	556	1 105	312	224	709	257	396	206	967	385	244	272	309	294	292	299

Table 9: Comparison on all key bytes for eShard

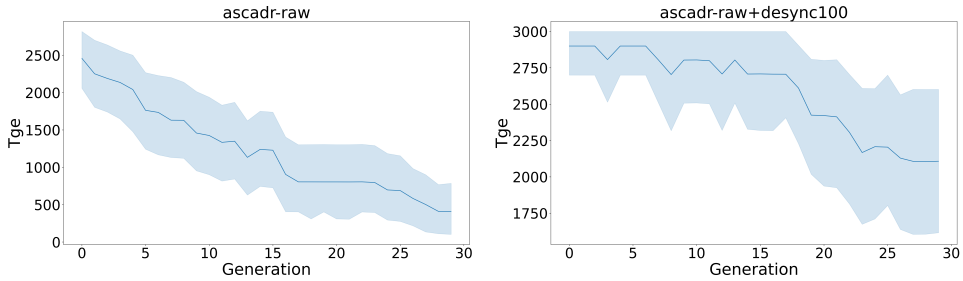


Figure 3: Convergence plots of the $T_{GE=1}$ metric over the number of generations for ASCADr on *raw* trace with $\delta_{max} = 0$ and $\delta_{max} = 100$. The shaded area is a 95% confidence interval on the average $T_{GE=1}$.

To substantiate the above argument, Figure 3 shows the convergence of the average $T_{GE=1}$ score across all runs and generations of SCAGE as an example, under the two scenarios of ASCADr where the traces are synchronized and with desynchronization up to 100. The plots show how the evolutionary approach progressively brings the $T_{GE=1}$ of the best individual towards the minimum of 1. As expected, SCAGE requires more generations to converge when desynchronization is applied.

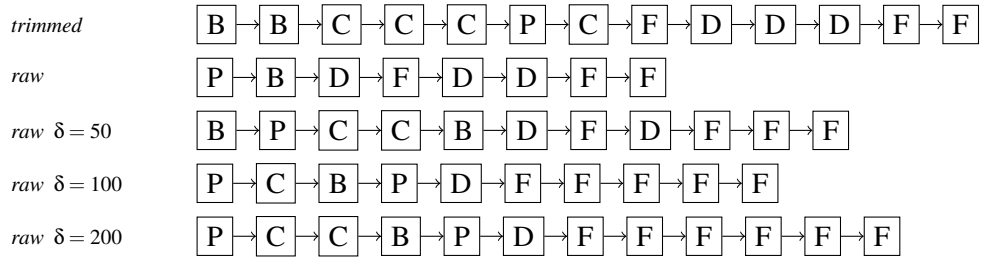
Efficient Search. Another strength of our tool is the ability to efficiently explore a large search space, both in terms of numbers of layers and hyperparameter ranges. Previous works that leveraged random search often had to significantly limit the possible ranges for its hyperparameters, as the resulting search space was already huge (e.g., more than 1 billion combinations in the setup of [26]). However, SCAGE does not suffer from this limitation, as the main advantage of evolutionary algorithms is to efficiently explore a large space by preserving the best components of the candidate solutions through selective pressure. Consequently, we can

Hyperparameter	[26]	SCAGE		
		Range	Max	Average
Convolution Layers	from 1 to 4	from 1 to 10	7	4.14
Convolution Filters	$4 \cdot 2^{i-1}$,$8 \cdot 2^{i-1}$,$12 \cdot 2^{i-1}$, $16 \cdot 2^{i-1}$ (i is the conv. layer index)	from 4 to 512	496	253
Convolution Kernel	26 to 52 with a step of 2	from 2 to 80	75	42.8
Pooling Size	2, 4, 6, 8, 10	from 2 to 20	20	11.6
Pooling Stride	same as Pooling Size	from 2 to 20	18	9.8
Dense Layers	from 1 to 4	from 1 to 6	6	3.42
Number of Neurons	10, 20, 50, 100, 200, 300, 400, 500	from 10 to 1 000	951	416

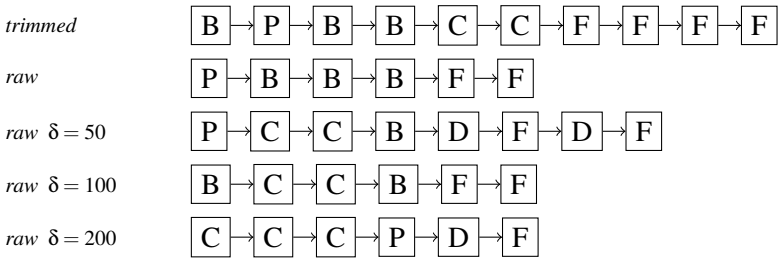
Table 10: Differences in the ranges of the search spaces explored by random search and SCAGE, and the maximum and average values actually attained by SCAGE.

run SCAGE on much larger intervals. Table 10 compares the ranges between the RS of [26] and those of SCAGE, together with the maximum and average values attained by the latter for the best NNs. The max and average values demonstrate that SCAGE actually uses values in the new region of the search space, suggesting that optimal architectures can be found outside of the usual ranges.

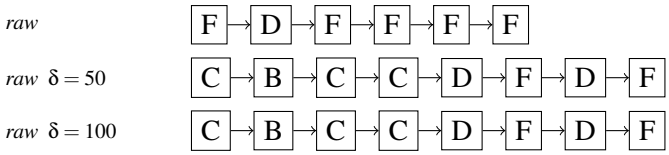
Architecture of the best models found. In Figure 4, we also analyzed the layer architecture of the best models found for ASCAD and CHES CTF 2018. For the comparison, we chose the models with the best $T_{GE=1}$ and, as a tie-breaker, picked those with the smallest number of trainable parameters. Regarding both ASCADf and ASCADr, we can immediately see how the best networks for the *trimmed* scenario have many more layers compared to their counterparts on *raw* traces. This difference, combined with the performance of SCAGE on the two scenarios, confirms the benefit of working with *raw* traces whenever possible, both in terms of $T_{GE=1}$ and final architecture. Another interesting remark concerns the preference of SCAGE for different types of layers depending on the underlying trace scenario. For the CHES CTF 2018 dataset, we intentionally evolved MLPs, but for ASCAD, we kept the usual setup with convolutional layers. However, the best models for ASCADf and ASCADr do not perform any convolution at all over the raw synchronized traces. On the other hand, convolutional layers always occur in the best networks evolved for trimmed synchronized and raw desynchronized traces. Therefore, SCAGE is capable of distinguishing the situations where the shift-invariance property of convolutional layers helps in circumventing the desynchronization countermeasure and where it can be safely omitted. A third interesting finding is on the best networks for the CHES CTF 2018 dataset with desynchronized traces, where SCAGE found the same exact network topology to be effective with $\delta_{max} = 50$ and $\delta_{max} = 100$, although it uses slightly different hyperparameters.



(a) ASCADf NN architectures



(b) ASCADr NN architectures



(c) CHES CTF 2018 NN architectures

Figure 4: Architectures of some of the best neural networks discovered by SCAGE in various scenarios. C stands for convolutional layer, P for pooling, B for batch normalization, F for fully connected, and D for dropout.

For eShard, we could not provide this type of comparison, having only the *trimmed* traces scenario. However, we analyzed the architectures of the best models obtained on every byte of the key, and we observed some common patterns. First, the number of convolutional layers remained well under the maximum of 10; in fact, the average was 4.7 layers, with a maximum of 6 used. Investigating the specific types of layers, on average, there are between 2 and 3 convolutions, 1 batch normalization, and 1 pooling layer. Therefore, it appears that, for this dataset, SCAGE naturally evolves neural networks without many nested convolution operations. The number of dense layers was also limited, with an average of 2 layers per network.

Comparison with other automatic hyperparameter tuning frameworks. Thanks to the unconventional combinations of layers, SCAGE can find models with competitive performance on many datasets. This makes it the most complete automatic hyperparameter tuning framework available for side-channel analysis, as the re-

Dataset	δ	Best Models' T_{GE1}					
		[29]	[38]	[2]	[31]	SCAGE on *trimmed* traces	SCAGE on *raw* traces
ASCADf	0	202	120	130	314	70	1
	50	443	-	-	531	309	3
	100	-	-	-	-	282	4
ASCADr	0	490	2945	120	-	23	1
	50	-	-	880	-	261	1
	100	-	-	960	-	x	1

Table 11: SCAGE vs. other automated search techniques.

sults collected in Table 11 demonstrate. All the considered methods are tested against the *trimmed* version of ASCAD, even in the case of desynchronization. Hence, we ran SCAGE also on this combination, and it also obtained the best result overall, unable to discover a successful model only for the ASCADr dataset with $\delta_{max} = 100$. In particular, it can be observed that SCAGE outperforms InfoNEAT in all scenarios except the one with the highest desynchronization level of 100. We hypothesize that this is due to the combined difficulty of considering both trimmed and highly desynchronized traces. However, we speculate that SCAGE could also break this scenario by further tailoring the grammar or tuning the parameters of the evolutionary strategy.

Execution time. Beyond the bare performance of the framework, there are some further considerations about the time required to run SCAGE. To provide a reference, the average times required to complete one evolution run of SCAGE (i.e., generate 5 offspring models for 30 generations) on the *raw* traces of ASCADf, ASCADr, and CHES CTF 2018 are, respectively, 3.5 hours, 14.0 hours, and 3.3 hours. For ASCAD with *trimmed* traces, the average times are halved. On eS-hard, the average time is between 2 and 3 hours. Using our HPC cluster, we could schedule up to 8 runs in parallel, which brings the total time to approximately 3 times the number of hours required for one run. For a high-level comparison, InfoNEAT [2] reports 2 days to complete an experiment, RL-SCA [29] states 100 hours, AutoSCA [38] 10 hours, and NASCTY [31] uses between 4 and 7 days. Since the experimental setup is not the same for every tool, it is hard to make a precise comparison. Nonetheless, these numbers confirm that SCAGE is aligned with previous works regarding the time required for the execution.

The difference is more noticeable in the execution times of the random search. The average time required to train a model is in the order of minutes. However, the main point in using a tool like SCAGE is the much higher chance of finding successful models, especially in harder scenarios where the success rate of a random search drops dramatically.

7 Conclusions and Future Work

Deep learning has proved to be a very effective tool to break implementations of cryptographic algorithms. Still, choosing an appropriate model is not an easy task, and researchers default to a random search when performing profiling attacks. In this paper, we showed the limits of RS and proposed an evolutionary approach based on grammar to mitigate them. Our tool, SCAGE, was extensively tested on common SCA datasets, using different feature selection strategies and increasingly higher levels of desynchronization. Our results demonstrate that SCAGE discovers successful SCA models much more reliably than a random search, especially in harder setups, where the chances of finding a good architecture drop significantly for RS. The best neural networks found in our experiments outmatch the performance of other state-of-the-art models in most of the considered settings. A further analysis shows that SCAGE takes full advantage of the freedom granted by the grammar we designed by developing unconventional architectures and hyperparameter combinations.

SCAGE mainly targets CNNs and can work with MLPs as well. In future work, we plan to extend the supported models to include new types of layers, such as those typical of transformer networks or recurrent neural networks. In our experiments, we showed that CNNs can still cope with a high level of desynchronization. However, this might not hold for other countermeasures, and different architectures may have to be considered.

References

[1] I. J. S. 27. Iso/iec 15408-1:2022: Information security, cybersecurity and privacy protection - evaluation criteria for it security, 2022.

[2] R. Y. Acharya, F. Ganji, and D. Forte. Information theory-based evolution of neural networks for side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 401–437, 2022.

[3] F. Assunçao, N. Lourenço, P. Machado, and B. Ribeiro. Towards the evolution of multi-layered neural networks: a dynamic structured grammatical evolution approach. In *Proceedings of the genetic and evolutionary computation conference*, pages 393–400, 2017.

[4] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro. Denser: deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines*, 20:5–35, 2019.

[5] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro. Fast denser: Efficient deep neuroevolution. In *European Conference on Genetic Programming*, pages 197–212. Springer, 2019.

[6] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro. Fast-denser++: Evolving fully-trained deep artificial neural networks. *arXiv preprint arXiv:1905.02969*, 2019.

[7] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas. Deep learning for side-channel analysis and introduction to ascad database. *Journal of Cryptographic Engineering*, 10(2):163–188, 2020.

[8] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[9] Federal Office for Information Security (BSI). Guidelines for Evaluating Machine-Learning based Side-Channel Attack Resistance. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_AI_guide.pdf?__blob=publicationFile&v=6`, 02 2024. Technical Report AIS 46.

[10] E. Galván and P. Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2(6):476–493, 2021.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] S. Hajra, S. Chowdhury, and D. Mukhopadhyay. EstraNet: An Efficient Shift-Invariant Transformer Network for Side-Channel Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):336–374, Dec. 2023.

[13] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to Biology, control, and Artificial Intelligence*. MIT Press, 1992.

[14] S. Karayalcin, M. Krcek, and S. Picek. A practical tutorial on deep learning-based side-channel analysis. Cryptology ePrint Archive, Paper 2025/471, 2025.

[15] S. Karayalçın, M. Krček, L. Wu, S. Picek, and G. Perin. It'sa kind of magic: a novel conditional gan framework for efficient profiling side-channel analysis. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 99–131. Springer, 2024.

[16] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):148–179, May 2019.

[17] K. Knežević, J. Fulir, D. Jakobović, S. Picek, and M. Đurasević. Neurosca: Evolving activation functions for side-channel analysis. *IEEE Access*, 11:284–299, 2023.

[18] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

[19] P. Kulkarni, V. Verneuil, S. Picek, and L. Batina. Order vs. chaos: A language model approach for side-channel attacks. *Cryptology ePrint Archive, Paper 2023/16151*, 2023.

[20] H. Li and G. Perin. A systematic study of data augmentation for protected aes implementations. *Journal of Cryptographic Engineering*, 14(4):649–666, 2024.

[21] X. Lu, C. Zhang, P. Cao, D. Gu, and H. Lu. Pay Attention to Raw Traces: A Deep Learning Architecture for End-to-End Profiling Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 235–274, July 2021.

[22] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[23] H. Maghrebi, T. Portigliatti, and E. Prouff. Breaking cryptographic implementations using deep learning techniques. In *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6*, pages 3–26. Springer, 2016.

[24] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[25] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[26] G. Perin, L. Wu, and S. Picek. Exploring feature selection scenarios for deep learning-based side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):828–861, 2022.

[27] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni. The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 209–237, Nov. 2018.

[28] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina. Sok: Deep learning-based physical side-channel analysis. *ACM Computing Surveys*, 55(11):1–35, 2023.

[29] J. Rijsdijk, L. Wu, G. Perin, and S. Picek. Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):677–707, 2021.

[30] U. Rioja, L. Batina, J. L. Flores, and I. Armendariz. Auto-tune pois: Estimation of distribution algorithms for efficient side-channel analysis. *Computer Networks*, 198:108405, 2021.

[31] F. Schijlen, L. Wu, and L. Mariot. Nascty: Neuroevolution to attack side-channel leakages yielding convolutional neural networks. *Mathematics*, 11(12):2616, 2023.

[32] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[33] M. Staib and A. Moradi. Deep learning side-channel collision attack. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):422–444, Jun. 2023.

[34] F.-X. Standaert, T. G. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009.*, pages 443–461. Springer, 2009.

[35] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[36] B. Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):107–131, Feb. 2019.

[37] L. Wouters, V. Arribas, B. Gierlichs, and B. Preneel. Revisiting a methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020, Issue 3:147–168, 2020.

[38] L. Wu, G. Perin, and S. Picek. I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. *IEEE Transactions on Emerging Topics in Computing*, pages 546–557, 2022.

[39] L. Wu, G. Perin, and S. Picek. Weakly profiling side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):707–730, Nov. 2024.

[40] L. Wu, A. Rezaeezade, A. Ali-pour, G. Perin, and S. Picek. Leakage model-flexible deep learning-based side-channel analysis. *IACR Communications in Cryptology*, 1(3), 2024.

[41] L. Wu, S. Tiran, G. Perin, and S. Picek. Plaintext-based side-channel collision attack. *IACR Communications in Cryptology*, 1(3), 2024.

[42] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli. Methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–36, 2020.