IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 3 November 2022, accepted 21 December 2022, date of publication 23 December 2022, date of current version 2 January 2023.

Digital Object Identifier 10.1109/ACCESS.2022.3232064

RESEARCH ARTICLE

NeuroSCA: Evolving Activation Functions for Side-Channel Analysis

KARLO KNEŽEVIĆ[1], **(Member, IEEE), JURAJ FULIR**[1], **DOMAGOJ JAKOBOVIĆ**[1], **(Senior Member, IEEE), STJEPAN PICEK**[2,3], **(Senior Member, IEEE), AND MARKO ĐURASEVIĆ**[1], **(Member, IEEE)**

[1]Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
[2]Intelligent Systems, Delft University of Technology, 2628 CD Delft, The Netherlands
[3]Digital Security Group, Radboud University, 6525 XZ Nijmegen, The Netherlands

Corresponding author: Stjepan Picek (s.picek@tudelft.nl)

ABSTRACT The choice of activation functions can significantly impact the performance of neural networks. Due to an ever-increasing number of new activation functions being proposed in the literature, selecting the appropriate activation function becomes even more difficult. Consequently, many researchers approach this problem from a different angle, in which instead of selecting an existing activation function, an appropriate activation function is evolved for the problem at hand. In this paper, we demonstrate that evolutionary algorithms can evolve new activation functions for side-channel analysis (SCA), outperforming *ReLU* and other activation functions commonly applied to that problem. More specifically, we use Genetic Programming to define and explore candidate activation functions (neuroevolution) in the form of mathematical expressions that are gradually improved. Experiments with the ASCAD database show that this approach is highly effective compared to results obtained with standard activation functions and that it can match the state-of-the-art results from the literature. More precisely, the obtained results for the ASCAD fixed key dataset demonstrate that the evolved activation functions can improve the current state-of-the-art by achieving a guessing entropy of 287 for the Hamming weight model and 115 for the Identity leakage model, compared to 447 and 120 obtained in the literature.

INDEX TERMS Activation functions, side-channel analysis, genetic programming, neuroevolution.

I. INTRODUCTION

Modern digital systems often use cryptographic solutions that serve as the basis for security, trust, and privacy protocols. Such solutions (algorithms) can be mathematically secure, but poor implementation decisions can make them vulnerable to attackers. One common vulnerability is the side-channel leakage [1]. Side-channel leakage exploits various information leakage sources. Common examples of leakage are power [2] and electromagnetic (EM) emanation [3]. Researchers have proposed several side-channel analysis (SCA) approaches to exploit those leakages with a common SCA division into non-profiling and profiling attacks. Non-profiling attacks such as Simple Power Analysis (SPA) [4] or Differential Power Analysis (DPA) [2] require fewer assumptions but may require thousands of measurements (side-channel traces) to break a target, especially if it is protected with countermeasures. Profiling attacks are considered the strongest possible SCAs [5]. In such attacks, the attacker has full control over a clone device. Then, the attacker builds the complete profile of the device and uses this profile to target similar devices to recover the secret information.

Deep learning approaches represent a powerful (and more recent) option for profiling SCA. The results in recent years show the potential of neural networks, where architectures like multilayer perceptron (MLP) and convolutional neural networks (CNNs) can break targets protected with countermeasures [6], [7]. Nevertheless, finding high-performing neural network architectures is often not straightforward due

The associate editor coordinating the review of this manuscript and approving it for publication was Ramakrishnan Srinivasan.

to the many hyperparameters to consider. In the hyperparameter tuning phase, we can distinguish between two different approaches. The first approach considers various techniques to select the best performing hyperparameters [8], [9], [10], [11]. Common techniques include gradient descent, Bayesian hyperparameter optimization, reinforcement learning, and evolutionary algorithms [12], [13], [14]. As discussed in Section III this is a well-explored direction in profiling SCA with excellent results, especially when using reinforcement learning and Bayesian optimization. The second direction deals with the design of custom neural network elements. In the SCA context, this direction is not extensively explored and mainly consists of the (manual) design of custom loss functions. Interestingly, those works showed excellent performance of such custom loss functions opening a question of whether other custom neural network elements could also improve the side-channel attack performance.

This paper considers the direction of the automated design of activation functions to be used in profiling side-channel analysis. A well-defined activation function can be any nonlinear function transforming the output of a layer in a neural network. Only a small number of different activation functions are commonly used in modern neural network architectures. Standard examples are the Rectified Linear Unit $ReLU(x)$ since it is simple and effective, or $tanh(x)$ and $\sigma(x) = 1/(1 + e^{-x})$ when it is needed to restrict the activation value within a certain range, such as in recurrent neural networks for language modelling [15]. There have also been works developing new activation functions with specific properties. For example, Leaky ReLU [16] allows information to flow when $x < 0$, or *Softplus* being positive, monotonic, and smooth [17]. There are many hand-designed activation functions [18], but none has become as widely used as *ReLU*. Still, there is significant (untapped) potential in developing custom activation functions for neural network design.

While there is undoubtedly significant potential in the design of custom activation functions, this is also a difficult problem. One option could be that a human expert would manually design a custom activation function. Unfortunately, this requires significant knowledge about the problem and can still ''miss'' many interesting solutions as they would not look intuitive to a human expert. Even if resolving those issues, the manual design would likely be very time-consuming and, as such, not appropriate for many real-world applications. On the other hand, it is possible to use automated techniques to design good activation functions. Then, the main issue becomes how to define reasonable search space ranges (i.e., the shape that the activation function can assume) and the search process procedure (i.e., how will the activation function be designed).

Our work develops an evolutionary approach to evolve activation functions for side-channel analysis. We draw on recent results on deep learning architectures and ask whether it is possible to make deep learning-based SCA even more efficient when the activation functions in a neural network are optimized for a given problem (neural network architecture, side-channel leakage model, and dataset). More specifically, we use Genetic Programming (GP), where we represent the activation functions as syntactic trees and evolve custom expressions. Due to their complexity, the resulting functions are unlikely to be discovered manually, but they (surprisingly) perform well, outperforming conventional activation functions like *ReLU* on common side-channel measurements like those in the ASCAD database. As far as we know, this is the first time that neuroevolution has been used for SCA or that evolutionary algorithms have been used to develop activation functions for SCA.

The main contributions of this paper are:

1) We evolve novel activation functions used in multilayer perceptron and convolutional neural networks. Neural networks with these activation functions show better performance than existing related research. **This shows that the newly developed activation functions have their place in the future designs of neural network architectures for SCA.**
2) We substitute common activation functions with evolved activation functions in previously developed neural network topologies and show better network performance after this substitution. **This shows that optimization of the activation function has relevance even when considering already developed neural networks.**
3) We analyze the structure of the evolved activation functions and their derivations, as well as the frequency of nodes occurring in the evolved activation functions. **Our analysis shows that certain nodes have more influence when constructing new activation functions, and as such, should be favored during the evolution process**.

We consider experiments on two datasets, two leakage models, two neural network types, and a number of scenarios.

The remainder of this paper is organized as follows. Section II covers the necessary definitions and notions. Section III discusses related works. Section IV provides the details about the proposed methodology. Section V describes the considered datasets and parameters. Section VI presents the experimental results. Section VII provides a deeper analysis of the results, most notably on the evolved activation functions. Finally, Section VIII sums up the paper's key contributions and gives possible avenues for future work.

II. BACKGROUND

This section introduces the required background knowledge regarding side-channel analysis, artificial neural networks, and genetic programming.

A. NOTATION

We will use calligraphic letters ($\mathcal{X}$) to represent sets and the corresponding upper-case letters (X) to represent random variables and random vectors $\mathbf{X}$ over $\mathcal{X}$. The corresponding lower-case letters x and $\mathbf{x}$ represent realizations of X and $\mathbf{X}$, respectively. The key candidate is denoted as k, where $k \in \mathcal{K}$, while the correct key is denoted as k^*.

As commonly done in SCA, a dataset is a set of traces (side-channel measurements) $\mathbf{D}$ with each trace $\mathbf{x}_i$ associated with an input value (plaintext or ciphertext) $\mathbf{i}_i$ and a key $\mathbf{k}_i$. To access a specific trace or input value, we use the index i. The dataset consists of three parts: profiling set (size N), validation set (size V), and attack set (size Q). Finally, the neural networks' vector of learnable parameters is denoted as $\boldsymbol{\theta}$.

B. MACHINE LEARNING-BASED SCA

We consider a typical profiling side-channel analysis setting with two phases: training (profiling) and testing (attack). A powerful attacker controls a device (commonly denoted as a clone device) with knowledge about the secret key. The attacker can obtain a set of N profiling traces, build a model f, and then use that model to obtain the secret key from the attack set of size Q. The function f is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^n$, with n being the number of trainable parameters:

- The profiling phase aims to learn $\boldsymbol{\theta}$ minimizing the empirical risk represented by a loss function on a profiling set of size N.
- The attack phase aims to make predictions about the classes:

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

where the class y is derived from the secret key and input through a cryptographic function and a leakage model.

We consider an attack on a block cipher (the AES cipher) and perform the multi-class classification task. More precisely, we learn a function f that maps an input to the output ($f : \mathcal{X} \to Y$)) based on input-output pairs, where the number of classes c is determined by the leakage model. Based on the class predictions, we estimate the effort required to reveal the secret key k^*. More precisely, a common result of predicting with a model f on the attack set is a two-dimensional matrix P with dimensions equal to $Q \times c$. The cumulative sum $S(k)$ for any key byte candidate k is a log-likelihood distinguisher:

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

The value $\mathbf{p}_{i,y}$ denotes the probability that for a key k and specific input, the result is class y.

Finally, to estimate the effort required to break the secret key, it is necessary to use side-channel metrics where the usual choice is to use the guessing entropy (GE) [19] metric. An attack results in a key guessing vector $\mathbf{g} = [g_1, g_2, \ldots, g_{|\mathcal{K}|}]$ in decreasing order of probability given Q traces in the attack phase (g_1 being the most likely key candidate and $g_{|\mathcal{K}|}$ the least likely key candidate). Guessing entropy is the average position of k^* in $\mathbf{g}$. In our work, we only consider attacks on specific key bytes, which is properly denoted as the partial guessing entropy metric. For simplicity, we nevertheless refer to it as guessing entropy, and we mention that it is a common assumption that all key bytes require similar effort to be guessed.

C. ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (ANNs) is a notion for all computer systems loosely based on biological neural networks. Such systems can "learn" from examples, making them a very popular paradigm in machine learning. Each ANN consists of a number of nodes, called artificial neurons, interconnected to transmit a signal. A simple type of neural network is called a perceptron. A perceptron is a linear binary classifier applied to the feature vector as a function deciding whether an input belongs to a categorical class or not. Each vector component has an associated weight w_i, and each perceptron has a threshold value q. The output of a perceptron is equal to "1" if the direct sum between the feature vector and the weight vector is greater than the threshold and "-1" otherwise. A perceptron classifier works only on linearly separable data, that is, when there is a hyperplane separating all positive points from all negative points [20].

1) MULTILAYER PERCEPTRON

We obtain a multilayer perceptron model by integrating more layers into a perceptron. The multilayer perceptron (MLP) is a feedforward neural network mapping input sentences to corresponding output sentences. Unlike the linear perceptron, MLP can distinguish data that is not linearly separable. MLP consists of multiple layers of nodes in a directed graph. Each layer is connected to the next, and each node in a layer is connected with a specific weight w to each node in the following layer. The multilayer perceptron model consists of at least three layers: an input layer, an output layer, and a hidden layer. These layers must consist of nonlinear activating nodes [21]. Backpropagation is used to train the network, where the gradient descent optimization algorithm uses backpropagation to adjust the weights of neurons by computing the gradient of the loss function [20].

2) CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) are neural networks initially developed for 2-dimensional convolutions, inspired by the biological processes of animals' visual cortex [22]. From an operational point of view, CNNs resemble ordinary neural networks (e.g., multilayer perceptron) [23]. CNNs use three main types of layers: convolutional layers, pooling layers, and fully-connected layers. Convolutional layers are linear layers that distribute weights across space. Pooling layers are nonlinear layers that minimize spatial size to inhibit the number of neurons. Fully-connected layers are layers where each neuron is connected to all neurons in the neighboring layer.

3) ACTIVATION FUNCTIONS

An activation function of a node is a function defining the output of a node as a function of an input or a set of inputs from a layer of linear nodes, as given in Eq. (2). To enable replication of nontrivial functions with ANNs that use a small number of nodes, one requires nonlinear activation functions:

$$y = activation\left(\sum_{i=1}^{|inputs|}(weight_i \cdot input_i) + bias\right). \tag{2}$$

Changes in the bias value allow the activation function to shift over the input domain, while changes in the weights alter the steepness of the activation function.

There are three types of activation functions: binary step function, linear activation function, and nonlinear activation functions. The problem with the first one is that it does not allow multi-value outputs, preventing multi-classification. A linear activation function suffers from two major problems: it cannot use gradient descent to train the model because the function's derivative is a constant, and all layers of the neural network collapse into one. A linear combination of linear functions is still a linear function, and such an activation function turns the neural network into a single layer [23].

Consequently, modern neural network models use nonlinear activation functions. They allow the model to create complex mappings between the inputs and outputs of the network, an essential part when learning and modeling complex data or data with high dimensionality. Nonlinear functions solve the problems of linear activation functions. First, they enable backpropagation learning as they have a derivative function related to the inputs. Finally, they allow the stacking of multiple layers of neurons to create a deep neural network.

There is no clear rule for selecting an activation function. In practice, activation functions are selected based on empirical results and execution speed. Some functions have theoretically justified properties, but that does not make them the best in practice. For example, *ReLU* is often used, and *tanh* is usually chosen for the hidden layers of recursive models, see, e.g., [24].

D. GENETIC PROGRAMMING

Genetic programming (GP) is an automated optimization method for developing computer programs used to solve complex problems in computer science [25]. The concept is based on general ideas derived from genetic algorithm theory and other evolutionary methods. Simply put, the ultimate goal of GP (as a product) is a general-purpose computer program that finds solutions to problems described using only input data and the desired results.

Although GP can be applied in many different ways and from many different perspectives, it represents a computer program as a syntactic tree in most cases. Indeed, any computer program can be represented as a tree or a forest of trees (in the broader sense), with the internal nodes of the tree having the role of operators (or functions of some number of variables) and the leaves having the role of operands. The set of operators (the function set) and operands (the terminal set) are predefined for a particular task. Evolution usually starts with a population consisting of randomly generated candidate solutions. These are called individuals whose operators and operands can be changed by three bio-inspired operations: selection, crossover, and mutation.

Selection is a process in which certain individuals are chosen from the current generation according to their fitness to serve as parents for the next generation. Individuals are selected probabilistically so that the better-performing individuals have a higher chance of being selected. In contrast, the crossover is a binary operator that exchanges information between two individuals to form a new offspring. Similarly, the mutation is a stochastic operator that helps increase the population's diversity by randomly selecting and changing one or more nodes in an offspring.

Then, in an iterative process, where one iteration is called a generation, the fitness of each new individual is evaluated. Based on their fitness, we select parent solutions for crossover. We then apply the crossover operator to pairs of individuals to generate new pairs of offspring, which are then additionally mutated with a given probability. The process is repeated until a predefined number of generations or other termination criteria are met.

III. RELATED WORK

A large body of research in SCA has addressed hyperparameter tuning but using general options (i.e., without developing custom neural network elements).

In 2016, Maghrebi et al. introduced convolutional neural networks for profiling SCA [26]. The authors also reported that they used genetic algorithms to tune the hyperparameters. While it is difficult to say whether this was the first time deep learning was used in SCA (many papers omitted details about neural network architectures), this work represented a significant turning point in SCA research and the first work using convolutional neural networks.

Cagli et al. were the first to propose using data augmentation for deep learning-based SCA [27]. They showed how such a technique could be used to improve the attack performance so that even an implementation protected with jitter can be broken. Kim et al. discussed a single neural network architecture design principle (VGG-like) that can be applied to multiple datasets [6]. Additionally, they showed how adding noise to inputs can improve the attack performance due to the noise regularization effect. Benadjila et al. introduced the ASCAD dataset, which now represents a standard benchmark for deep learning SCA [53]. Additionally, they provided the first well-performing architecture for that dataset by evaluating several CNN hyperparameters. Perin et al. provided insights into why output class probabilities represent a strong metric for SCA [28]. Furthermore, they used random search in predefined domains to build neural networks to form ensembles. Some of the ensemble results still represent state-of-the-art attacks. Rijsdijk et al. investigated how reinforcement learning can be used to tune hyperparameters for CNNs [29]. They reported excellent results (attack performance with small neural network architectures) but at the cost of significant computational resources to find such neural networks. Wu et al. used Bayesian optimization for hyperparameters tuning for MLP and CNN architectures [30]. The obtained results were small architectures that efficiently break targets, requiring a fraction of the cost compared to the reinforcement learning approach.

Zaid et al. proposed a more structured approach for selecting hyperparameters determining the size of layers in CNNs [7]. The authors considered the number of filters, kernel sizes, strides, and the number of neurons in a fully connected layer and found small neural networks for several datasets that significantly reduced the attack effort. Wouters et al. [31] improved on the work of Zaid et al. [7] and showed comparable attack performance while requiring neural network architectures with even less trainable parameters. Acharya et al. used neuroevolution to evolve architectures (thus, similar to [26]) but also neural network parameters [32]. Still, the authors considered only "standard" elements in the hyperparameter evolution and did not evolve custom ones. More recently, the SCA community turned its attention to deep learning-based SCA with no feature engineering [33], [34] and showed the benefits of using the extra information stemming from more features.

Finally, several works focus on improving the performance of deep learning-based SCA by developing custom neural network elements. Pfeifer and Haddad designed a new type of layer called "Spread" and claimed that it reduces the number of layers required and accelerates the learning phase [35]. Zaid et al. introduced a new loss function derived from the rank learning approach, which helps to avoid approximation and estimation errors [36]. Zheng et al. proposed a new metric function called Cross Entropy Ratio (CER), which they adapted to a new loss function specifically designed for deep learning in SCA [37]. Kerkhof et al. proposed a loss function that is designed for SCA, performs well, and has low computational overhead [38]. Finally, the same authors also conducted a systematic analysis of custom loss functions and commonly used ones and concluded that custom loss functions indeed perform better in SCA [39].

On the other hand, many research works deal with the problem of obtaining the optimal architecture and hyperparameters of artificial neural networks using various methods such as Bayesian optimization [40] or network pruning [41]. One specific part of this field is the evolution of activation functions using evolutionary algorithms and machine learning. A first attempt to learn activation functions in a neural network is found in [42], where the authors proposed to randomly add or remove logistic or Gaussian activation functions using GP. In [43], the authors developed a method to automatically select an activation function for each neural network layer. Hagg et al. extended the NEAT algorithm [44] to simultaneously develop activation functions per neuron in addition to the entire network topology [45]. Both works used predefined lists to select activation functions. Ramachandran et al. used reinforcement learning to automatically design activation functions [46]. In doing so, the authors discovered a number of powerful new activation functions and analyzed one in detail: $x \cdot \sigma(x)$, which they call Swish. Bingham et al. complemented previous research by investigating an evolutionary algorithm approach for developing activation functions [47]. The authors showed it is possible to develop specialized activation functions that perform well for the CIFAR-10 and CIFAR-100 datasets. In [48], the authors used a hybrid genetic algorithm to evolve a function that is defined differently for the positive and negative domains. Parts of the function are represented by trees and crossed by special operators that modify the positive and negative sides separately. The set of nodes consists of basic arithmetic operations, and the leaves are popular activation functions without constants. The authors also introduced the new activation functions ELiSH and Hard ELiSH, which they developed manually to combine the good properties of the smaller functions. Using three datasets, they showed that their functions performed best.

In [49], the authors represent activation functions as expression trees and use standard evolutionary algorithms to evolve new activation functions. Their results demonstrate that by replacing the standard ReLU activation function with these newly evolved functions improves the performance of neural networks on the CIFAR datasets. A similar approach is also investigated in [50], where the authors apply genetic programming to search for new activation functions. Through their experimental analysis, the authors demonstrate that their new activation functions outperform several standard functions on multivariate classification problems. Since in both previous studies GP is used to evolve new activation functions, it is possible to obtain functions of various forms. However, in some cases this can also lead to the generation of quite complex activation functions which are difficult to interpret. Differential evolution is applied in [51] for evolving new activation functions for the long-short-term memory networks. The proposed method builds a hierarchical activation function of a predefined structure by searching for the most appropriate function elements that should appear in it to represent a complete activation function. Although the proposed method can construct activation functions that outperform traditional activation functions, like ReLU, due to the way in which solutions are represented, the method can only construct activation functions of a given structure. A coevolutionary algorithm to evolve new activation functions for standard fully-connected and convolutional neural networks is proposed in [52]. The authors use Cartesian GP to evolve new activation functions, which can evolve new activation functions of various structures. However, similar to when GP was used, Cartesian GP is also prone to generate quite complex activation functions. An extensive experimental benchmark on the MINST and similar datasets outlines the proposed approach to be suitable for finding new activation functions. From the previous studies dealing with evolution of activation functions, we see that different approaches were tested, some which allow more freedom in the design of activation functions (GP and Cartesian GP) and others which restrict their structure (differential evolution), but in both cases the newly generated activation functions outperform standard ones.

IV. METHODOLOGY

The process of selecting an appropriate activation function for a given problem is time-consuming. Furthermore, there is no guarantee that the set of candidate activation functions even contains the function that would be the most appropriate for the given problem. As a consequence, a suboptimal activation function might be selected to be used for solving a given problem. A possible remedy for this would be to design an appropriate activation function for the problem at hand. Naturally, doing this manually would be impossible, not only due to the sheer number of problems for which this would have to be done but also because the design of activation functions is not a straightforward and easy task. Therefore, it makes sense to automatize this process to perform an automatic design of activation functions for the considered problem.

To automatically design appropriate activation functions for side-channel analysis, we apply GP to evolve novel activation functions. The procedure is performed in two phases as outlined in Figure 1:

1) optimization of neural network parameters,
2) evolution of a new activation function.

In the first phase, we use architecture search to find the representative architecture for each dataset (and neural network type). Two different algorithms were used to explore the space of network architectures and their hyperparameters: grid search and random search. In both techniques, a large number of points in the search space[1] are evaluated to find the optimal point. Grid search evaluates all possible combinations of parameter values for a given search space, sampling the continuous variables with fixed steps. On the other hand, the random search strategy samples the given space by randomly selecting points in the search space. The points are evaluated on the test set to obtain a distribution of representative solutions, which are later used for further optimization with evolution. Therefore, in the first phase, a set of parameter values is generated in each iteration and using the sampled parameter values, training of the ANN is performed. This is repeated until the search is finished (either by examining all or a predefined number of combinations). The parameter set for which the best results were obtained is returned as the final result.

After all the relevant parameters of the ANN have been selected in the first phase, an appropriate activation function for that set of parameters is evolved using GP, which constitutes the second phase of the proposed methodology. Each activation function in the search space represents a feasible solution and is represented as a tree consisting of unary and binary operators whose leaves correspond to the function inputs $\vec{x}$, i.e., the outputs of a dense linear layer. We consider the following operators:

- Unary: $\vec{x}$, $-\vec{x}$, $|\vec{x}|$, $sin(\vec{x})$, $cos(\vec{x})$, $e^{\vec{x}}$, $erf(\vec{x})$, $\vec{x}^2$, $1//\vec{x}$, $\sigma(\vec{x})$, $\sigma_H(\vec{x})$, $ReLU(0, \vec{x})$, $ELU(\vec{x})$, $Softsign(\vec{x})$, $Softplus(\vec{x})$, $tanh(\vec{x})$.
- Unary, multidimensional: $normalized(\vec{x})$, $Softmax(\vec{x})$, $Softmin(\vec{x})$.
- Binary: $\vec{x_1} + \vec{x_2}$, $\vec{x_1} - \vec{x_2}$, $\vec{x_1} \cdot \vec{x_2}$, $\vec{x_1} // \vec{x_2}$.

The operator $//$ denotes the protected division in which the value of the denominator is replaced by $\epsilon = 10^{-4}$ if the absolute value of the denominator is less than ϵ. *ReLU* denotes the rectified linear unit, *ELU* the exponential linear unit, *erf* the Gaussian error function, σ the sigmoid function, σ_H the hard sigmoid function, and *normalized* the L_2 vector normalization. The initial tree depth is between 2 and 5, and the maximum tree depth is limited to 12. Moreover, there is no restriction on tree balancedness as discussed in [46] and [47].

GP iteratively performs standard genetic operators on a given population of solutions. The GP algorithm uses the steady-state tournament selection, where the size of the tournament (k) is equal to 3. Starting from a population of P randomly generated individuals representing activation functions, a tournament with three randomly selected individuals is conducted in each iteration. The worst individual from the tournament is eliminated, and a new individual is formed by performing a crossover on the remaining two individuals. For crossover, the simple tree crossover operator is used, in which a random node, denoted as a crossover point, is selected in each parent. The subtrees rooted in those nodes are then swapped to produce two new child individuals. The crossover is performed with a 90% bias for functional nodes being selected as crossover points to stimulate the transfer of a larger amount of genetic material. The mutation is applied to the new individual, subject to the individual mutation rate. In mutation, a node in an activation function tree is selected uniformly at random. At this point, the selected subtree is replaced by generating a new subtree, respecting the maximum depth limit. With P new individuals generated in this way, a single generation of the evolutionary process is completed. The best activation function is always retained in the population (elitism) since it can never be selected for elimination. We run the process for a number of generations, and the activation functions with the best performance are returned as a result. The outline of the applied GP is given in Algorithm 1.

Algorithm 1 Steady-State Tournament GP

```
randomly select $k$ individuals;
remove the worst of $k$ individuals;
$child$ = crossover (best two of the tournament);
perform mutation on $child$, with given individual mutation probability;
insert $child$ into population;
```

Each time an individual needs to be evaluated during the run of GP, the activation function that the individual represents is embedded into the neural network with the parameters

[1] The space of all feasible solutions is called the search space.

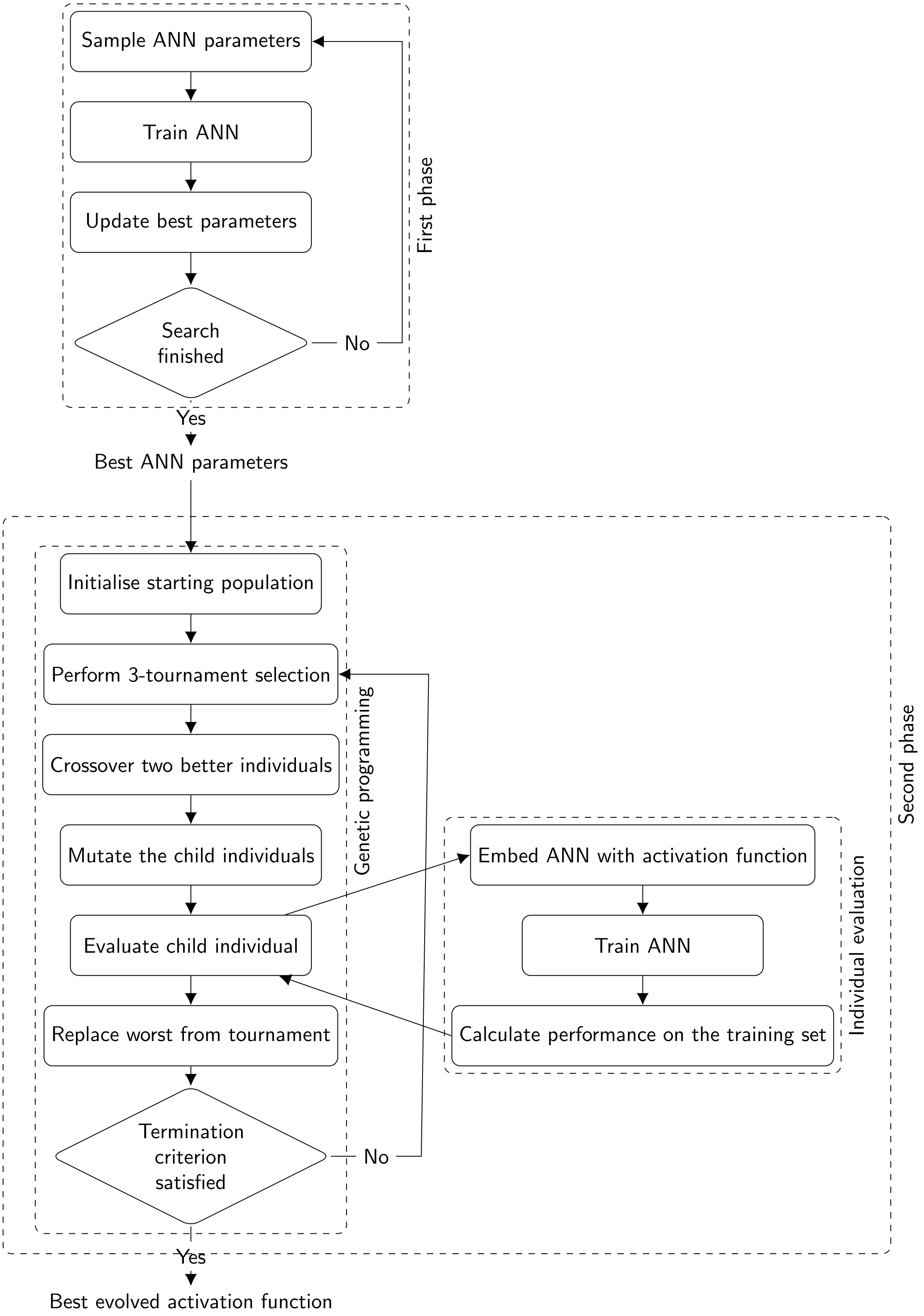

Sample ANN parameters
Train ANN
Update best parameters
Search finished
No
First phase
Yes
Best ANN parameters
Initialise starting population
Perform 3-tournament selection
Crossover two better individuals
Mutate the child individuals
Evaluate child individual
Replace worst from tournament
Termination criterion satisfied
No
Genetic programming
Embed ANN with activation function
Train ANN
Calculate performance on the training set
Individual evaluation
Second phase
Yes
Best evolved activation function

FIGURE 1. Flowchart of the proposed methodology.

obtained from phase 1. The network is then trained on the training set for a given amount of time, after which its performance on the training set is used to denote the fitness (quality) of the individual. Since the ANN needs to be trained in each evaluation of an individual, this represents the most computationally expensive part of the GP procedure. After the specified termination criterion is achieved, the GP algorithm returns the best individual in the population as the final result, representing the best activation function obtained during the evolutionary process. Therefore, the complexity of the proposed method depends on two parameters, the termination criterion used when training the ANNs, which specifies the amount of time allocated to each network for training, but also the termination criterion used by GP, which influences how many ANNs will be evaluated during the evolution process. As such, the run-time of the proposed method can be controlled by adjusting the values of these two parameters.

Naturally, it could be said that some architectures are not selected in the architecture search but would be with custom activation functions, advocating the need for simultaneous architecture search and neuroevolution process. Still, we decided not to follow such a path as the computational complexity of considering both neuroevolution and architecture search simultaneously would be extreme.

V. EXPERIMENTAL SETUP

This section provides detailed information on the setup of the experiments, outlining the data sets used, evaluation criteria, and selected parameter values for the considered methods.

A. DATASETS

We use two versions of the ASCAD database [53], representing commonly used datasets for deep learning-based SCA evaluation. This database contains the measurements from an 8-bit AVR microcontroller running a masked AES-128 implementation.[2]

The first version of the ASCAD database has a single (fixed) key. This dataset has 50 000 traces for profiling and 10 000 for the attack. The traces in this dataset have 700 features (preselected window when attacking the third key byte). We consider the first 45 000 traces for training and 5 000 for validation from the original training set.

The second version of the ASCAD database has random keys for the training set, while the key is fixed for the test set. The dataset consists of 200 000 traces for training and 100 000 for the attack. Each trace in this database has 1 400 features (preselected window for the third key byte). We use 5 000 traces from the original training set for validation. We normalize the input features for both data sets to a Gaussian distribution (zero mean and unit variance) by computing the distribution parameters of the training set.

For both datasets, we attack the third key byte only, as this is the first masked key byte. It is a standard practice to attack only one key byte as it is expected that the attack difficulty should be similar for the other key bytes, see, e.g., [7], [31]. The preselected windows are chosen to contain points of interest around the leaking spot in the first round. Again, this represents a common practice used in related works we consider when comparing the results. We note that more recent works avoid this practice and consider the raw traces, but the comparison with such works would not be fair and is thus out of the scope of this work.

B. LEAKAGE MODELS

We evaluate the performance of deep learning-based SCA for two leakage models: the Hamming Weight (HW) and Identity (ID). In the HW leakage model, the attacker works under the assumption that the leakage is proportional to the Hamming weight of the sensitive variable. This leakage model results in nine classes for a cipher that uses an 8-bit S-box, as is the case for the AES cipher. Since this leads to a strong imbalance in the label distribution, we also compute the imbalance weights that balance the computed model loss. We follow the guidelines for calculating the imbalance weights from [54]. For the ID leakage model, the attacker assumes the leakage proportional to an intermediate value of the cipher. When considering an 8-bit S-box, this leakage model results in 256 classes (values between 0 and 255).

C. ARCHITECTURE SEARCH STRATEGIES

We consider two types of neural networks: CNN and MLP. To compare our results, we will consider several state-of-the-art architectures [7], [30]. We note that not all results are directly comparable due to certain differences in training/validation/test set sizes, but we are confident some general observations can be made. The rest of the section details the grid and random search methods used to find appropriate ANN parameters.

1) GRID ARCHITECTURE SEARCH FOR CNNs

We consider grid search for CNNs because the number of hyperparameters is very large, making the random search more difficult. Due to the time complexity, we have restricted this search space by removing hyperparameter values that are not expected to yield good results (e.g., very shallow or narrow architectures). The hyperparameter space considered is described in Table 1 and yielded 2 160 grid samples to obtain the best convolutional model (CNN - GS_{best}) for each of the datasets.

2) RANDOM ARCHITECTURE SEARCH FOR MLP

For MLP, we use random search since related work has achieved good results even with such a simple tuning setup [28], [30]. It is similar to grid search but without a structured sampling of continuous spaces, leading to bias since only a subset of the possible values is selected. We consider this bias undesirable because we do not use a learning rate schedule for our MLP models, and the model could be more sensitive to the exact value of a fixed learning rate. We define the search space as the collection of hyperparameters that affect the shape of the MLP architecture and its train parameters, listed in Table 1. The search space differs slightly from the grid search strategy in that it provides higher resolution for the layer widths, learning rate, and over multiple seed values to compensate for the

[2] Publicly available at https://github.com/ANSSI-FR/ASCAD/

TABLE 1. Definitions of the architecture search spaces. The values in square brackets represent a continuous range, while the curly brackets denote a set of discrete values.

Parameter	Grid search subspace	Random search subspace
Seed	36	[0,100]
Number of layers	{2, 3, 4}	[2, 8]
Layer width	{10, 15, 20, 25, 100}	[100, 1 000]
Learning rate	5e-3	[1e-4, 1e-2]
Optimizer	{SGD, RMSProp, Adam}	{SGD, RMSProp, Adam}
Activation function	{ReLU, ELU, SELU, tanh BLU APL GELU}	{ReLU, ELU, SELU, tanh, Sin, LReLU BLU APL GELU}
Train epochs	{20, 25, 50, 75}	50

possibility of poor initialization. We sample 600 random points from this space to obtain an approximate distribution of the actual solution space for MLP architectures for each dataset. From this, we can determine the best model (MLP - RS_{best}) and the median model (MLP - RS_{median}). We also use median models to better assess the average performance, as outliers are highly likely with random search.

D. EVOLVING ACTIVATION FUNCTIONS

1) GP PARAMETER VALUES

The search space of functions is large and requires some assumptions to make the search feasible. First, we restrict the leaves to the function's input, with no constants or learnable parameters. Next, we limit the maximum depth of the candidate trees to restrict the search to a subspace of functions that can be evaluated efficiently since they must be evaluated for each layer. Finally, we limit the expressiveness of the representation by defining a set of possible unary and binary operations that can be used as function nodes (see Section IV). We use a population of 20 individuals and run the algorithm with a budget of 2 000 evaluations. The mutation probability was selected from the preliminary experiments with the ASCAD fixed key dataset and was kept at 70% in all experiments. These settings were selected after a tuning phase as they provide a good trade-off between the computational efficiency and the attack performance.

2) FITNESS FUNCTION

A neural network is trained with each fitness function on a given training dataset, starting with a population of P activation functions. Recall that guessing entropy denotes the average key rank, i.e., the correct key position in the guessing vector after processing Q attack traces. Thus, our goal is to minimize the guessing entropy for any number of attack traces. Therefore, it is natural to consider the number of attack traces necessary to reach GE of 0, which we refer to as $\overline{Q}_{tGE}$. Each candidate function is assigned a fitness value F:

$$F = \overline{Q}_{tGE} + (1 - accuracy). \tag{3}$$

The goal is to minimize the fitness value F, where the optimal value is 1 (this would require only a single trace to break the target, which is the optimal scenario). The guessing entropy is averaged over 100 attacks on randomly selected data subsets. The maximum size of the subsets was chosen depending on the experiment to balance differentiation in result quality and computation time. This, in turn, leads to similar results between individuals in the first iterations and slows down the convergence of GP. To remedy this, we add the accuracy error ($1 - accuracy$) to the fitness, which is also subject to minimization and provides additional information to differentiate between results. In initial experiments, we observed faster convergence using this fitness function.

We note that it may be somewhat counterintuitive to use accuracy for SCA. This problem may be particularly pronounced for the HW leakage model, as it leads to highly imbalanced data [54]. Since the second part of the fitness function is limited in the range [0, 1], additional information about the accuracy can be helpful for neural networks that perform well (i.e., those that reach GE of 0 in a small number of traces) by providing more slope in the search space. However, the contribution of the accuracy term is small, so the number of traces remains the primary objective.

E. LEARNING SYSTEM

All experiments were performed on a computer with a single GeForce GTX 1080 Ti graphics card, an i7-6700 CPU, and 32 GB of RAM running Ubuntu 16.04. We performed our experiments in Python 3.7 using the DEAP [55] framework (v1.3.1) for evolutionary algorithms and the PyTorch framework [56] (v1.7.0) for deep learning with the CUDA backend (v11.2). Since our model architectures do not require a large amount of GPU memory, we take full advantage of the hardware by parallelizing the evaluation steps of individuals via multiprocessing. This proved crucial since our experiments are time-consuming, in the order of seven days per architecture search and 14 days per evolution without parallelization.

VI. EXPERIMENTAL RESULTS

This section first outlines the results of a preliminary experiment in which different activation functions were tested, after which we present the results obtained when evolving new activation functions.

A. PRELIMINARY COMPARISON OF ACTIVATION FUNCTIONS

To demonstrate the variability in the results obtained with various activation functions, a preliminary series of experiments was performed in which the MLP and CNN models were trained with different parameters and using different activation functions. Different popular activation functions from the literature were selected to determine how they perform on the considered problem.

Table 2 represents preliminary results obtained for this series of experiments, in which the HW leakage model was considered. The results denoted in this table were obtained by performing the random and grid search procedures on the

TABLE 2. Best $\overline{Q}_{t_{GE}}$ values obtained by standard activation functions for the ASCAD datasets. The values in bold denote the best-achieved results.

Act. func.	Fixed key		Random keys	
	CNN	MLP	CNN	MLP
ELU	299	972	1 026	1 592
ReLU	523	801	>5 000	2 128
SeLU	323	**561**	606	2 078
Tanh	1 042	1 374	>5 000	2 575
PReLU	387	1 002	**470**	1 565
GELU	314	840	737	2 473
BLU	**257**	584	**470**	1 482
APL	270	761	541	**1 249**

parameter set outlined in Table 1. Thus, the result outlined for each activation function represents the best value obtained for any of the considered parameter combinations. We can clearly observe that there is not a single activation function that performs best across all of the scenarios that were considered. On the contrary, some activation functions that perform well in some scenarios perform quite poorly in others, as is the case with the SeLU activation function. This initial experiment outlines that no single activation function is appropriate for all situations and that selecting an inappropriate activation function can result in poor performance of the trained ANN.

B. EVOLUTION OF NOVEL ACTIVATION FUNCTIONS

This section presents experimental results showing that the developed activation functions can outperform the commonly used activation functions. We first search for the optimal network architecture and hyperparameters using the architecture search methods discussed previously (random search - RS and grid search - GS) for each experimental setup. The final hyperparameter values are listed in Table 3.

Then, we apply the evolutionary algorithm to further optimize the model by changing its activation function. Finally, we compare the results of the above techniques on both datasets and leakage models. While evaluating GS and RS on the ASCAD fixed key dataset with the ID leakage model, we limited the evaluation of $\overline{Q}_{t_{GE}}$ to 1 000, as we found that over 25% of the results were in this subspace, resulting in an efficient region of interest. Additionally, we focused on 500 traces when evaluating GP to improve efficiency further while still producing better results. For the random keys dataset, we had to increase the truncation bar to 1 000 as it became more difficult to obtain 25% of the results in this subspace. Finally, for all tasks in the HW leakage model, we increased the bar to 5 000 as they proved to be more difficult.

Based on the results obtained during the architecture tuning phase denoted in the previous section, we examined the performance of commonly used (ELU, SeLU, ReLU, Tanh, APL, GELU, BLU) activation functions. We observed that no single activation function was consistently dominant for all considered problem configurations, although in most cases, the best results were obtained when using the BLU activation function. We consider this to be an interesting result since, to the best of our knowledge, the BLU activation function was never before discussed in the context of SCA. Similarly, none of the activation functions was consistently the worst. This shows that the performance of the activation functions is highly dependent on the problem under consideration. Even SeLU, which generally performs the best, sometimes achieved quite bad results. Moreover, we observed that a change in the activation function could easily result in the attack performance requiring more than double the traces than before. This motivates us to search for more suitable activation functions for the considered problems.

To compare the obtained results with state-of-the-art, we consider a number of results (we report the number of attack traces to break the target, i.e., to reach a guessing entropy of 0):

- For ASCAD fixed key and the ID leakage model, we consider the results from [7]. There, with 45 000 training traces, the authors required 191 traces to break the target. Additionally, while the authors did not consider the HW leakage model, later research works adapted the architecture for the ID leakage model by adjusting the number of classes and reached 1 246 traces to break the target [30].
- For the ASCAD random keys, the authors in [28] used ensembles. They reported 470 required traces for the HW leakage model and 105 for the ID leakage model. The authors used 200 000 traces for the training set.
- In [30], the authors used Bayesian optimization to find well-performing architectures. For ASCAD fixed key, they needed 447 and 120 traces for the HW and ID leakage models, respectively. For ASCAD random keys, they required 496 and 2 945 traces for the HW and ID leakage models, respectively. The authors used a training set of 50 000 traces.
- In [29], Rijsdijk et al. used reinforcement learning to optimize the architectures. For ASCAD fixed key, they reported 906 and 202 traces needed to break the target for the HW and ID leakage models, respectively. For ASCAD random keys, the number of required attack traces was 911 and 490 for the HW and ID leakage models, respectively. The training set size equals 45 000 traces.
- In [38], Kerkhof et al. design custom loss functions for SCA. The training set size equals 50 000 traces, and the authors report median performance. Note that in this work, custom loss functions are used with the architectures obtained through random search. For ASCAD fixed key, the number of attack traces equals 490 and 520 for the HW and ID leakage models, respectively. For ASCAD random keys, the number of traces equals 800 and >3 000 for the HW and ID leakage models, respectively.

We emphasize that we do not differentiate between various architecture types (MLP or CNN) when reporting results from related work. Rather, we provide the best obtained values. We note that all the aforesaid values are obtained from the

TABLE 3. Selected hyperparameter values for the experiments.

	Fixed key				Random keys			
	HW		ID		HW		ID	
	CNN	MLP	CNN	MLP	CNN	MLP	CNN	MLP
Hidden layers	4	8	2	5	4	2	2	5
Layer width	100	478	10	661	100	716	100	622
Activation	ELU	ReLU	SELU	ELU	SELU	SELU	Tanh	ReLU
Optimiser	RMSProp	Adam	Adam	Adam	SGD	SGD	SGD	Adam
Learning rate	0.005	0.0017	0.005	0.0086	0.005	0.0042	0.005	0.00086
Batch size	64	200	50	200	50	200	64	200
Epochs	20	50	50	50	75	50	50	50

OPOI (optimized points of interest) setup as discussed in [34]. More precisely, the attack is conducted on the optimized interval of points of interest. This is a common setup that most of the related works follow and our experiments also follow that setup. There is also a NOPOI (non-optimized points of interest) setup that considers raw features and allow excellent attack performance, but we do not consider it here.

In Tables 4 and 5, we show the best results for the Hamming weight and the ID leakage models, respectively. The notation $> x$ means that we could not achieve GE of 0 with x attack traces. For neuroevolution, we provide the best three results to better indicate the variation due to the different evolved activation functions but also to indicate there are many evolved activation functions that perform well.

For the HW leakage model (Table 4) and the ASCAD fixed key dataset, the best results are obtained with an already existing activation function. Although GP could not attain the best result, in this case, it achieved the third-best result among all the tested activation functions. The random search results for MLP are much worse than the first two, but we still manage to break the target. At the same time, the results from related works reach a better performance than random search but worse than the results obtained here. For the random keys dataset, the best results for CNN are obtained with the grid search. The same performance is achieved in related works with ensembles and slightly worse with Bayesian optimization. Finally, the results for the custom loss function are almost twice worse. Interestingly, CNN constructed using GP fails to converge even with 5 000 attack traces. The architectures end up with the guessing entropy values: 108, 110, and 111, respectively, for the top 3 individuals. We suspect this is because the random keys dataset is more difficult and easier to overfit. Nevertheless, adding more generations to the GP procedure could improve its behavior and attack results.

Note that MLP with neuroevolution is evaluated separately later as we also consider the performance of median models.

In Table 5, we show the results for the ID leakage model. Interestingly, for the fixed key, we see that CNNs evolved with GP perform better than related work. The results with Bayesian optimization are rather similar to the second-best results with GP. Other related work results give worse results than our approach. For random keys, we obtain good results only with MLP with random search. Again, this confirms that the random keys setup is more difficult, and we need a more sophisticated search process to reach good results. The CNN - GS_{best} ends with a guessing entropy of 128, while the top 3 evolved results end with values: 125, 128, and 128, respectively. Slightly better results can be found in related works when using ensembles.

TABLE 4. Final $\overline{Q}_{t_{GE}}$ values for the ASCAD fixed and random keys datasets with the Hamming weight leakage model. We compare the best-obtained value of grid search on the CNN model (CNN - GS_{best}, with its evolved activation function CNN - GP and the best obtained MLP model on random search MLP - RS_{best}. The best values are denoted in bold style.

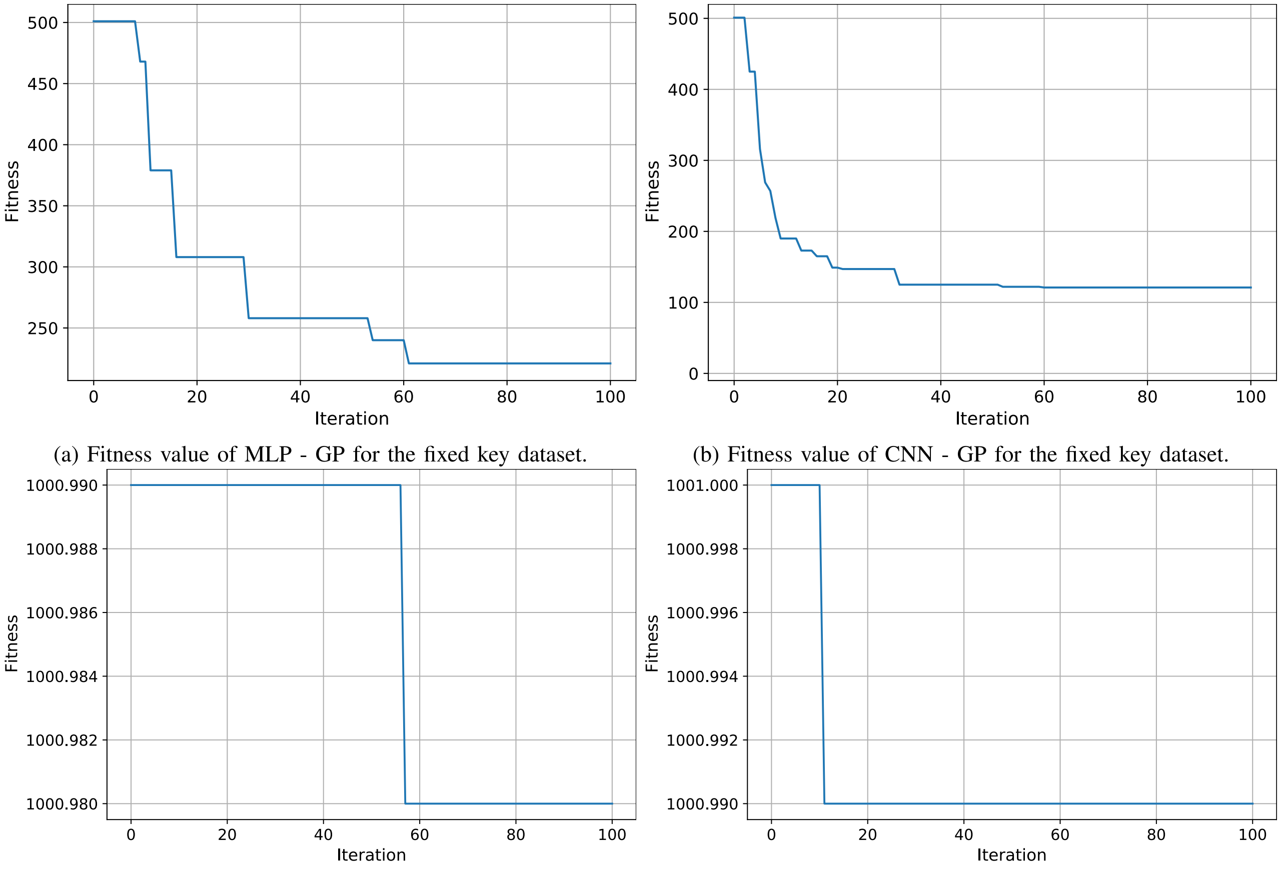
	Fixed Key			Random Keys		
	best result	2nd best	3rd best	best result	2nd best	3rd best
CNN - GS_{best}	**257**	-	-	**470**	-	-
CNN - GP	287	331	339	>5 000	>5 000	>5 000
MLP - RS_{best}	561	-	-	1 133	-	-

TABLE 5. Final $\overline{Q}_{t_{GE}}$ values for the ASCAD fixed and random keys datasets with the ID leakage model. We compare the best-obtained value of grid search on the CNN model (CNN - GS_{best}), the version with its evolved activation function (CNN - GP), and the best obtained MLP model on random search MLP - RS_{best}. The best values are denoted in bold style.

	Fixed Key			Random Keys		
	best result	2nd best	3rd best	best result	2nd best	3rd best
CNN - GS_{best}	191	-	-	> 1 000	-	-
CNN - GP	**115**	123	130	>1 000	>1 000	>1 000
MLP - RS_{best}	156	-	-	**145**	-	-

Next, in Table 6, we give the median results for the HW leakage model when comparing MLP architectures obtained with random search and after developing activation functions with GP. Note that GP significantly improves the performance for the fixed key scenario, while the GP approach does not converge for the random keys dataset. Since the GP results show we do not manage to break the target, this again indicates that we require more than 100 generations to develop good activation functions.

Finally, in Table 7, we compare the median results for MLP with random search and after using GP considering the ID leakage model. Observe how GP reaches significantly better results for both fixed key and random keys settings. This suggests that while the random search can find a powerful neural network architecture by simply guessing, the average results obtained over a number of solutions are not necessarily good. On the other hand, evolving customized activation functions can significantly improve the neural network's performance.



(a) Fitness value of MLP - GP for the fixed key dataset.

(b) Fitness value of CNN - GP for the fixed key dataset.

(c) Fitness value of MLP - GP for random keys dataset.

(d) Fitness value of CNN - GP for random keys dataset.

FIGURE 2. Evolution of fitness value for ASCAD with the ID leakage model.

TABLE 6. Results of the EA effectiveness experiment for ASCAD fixed and random keys with the Hamming weight leakage model. The median architecture is compared before MLP - RS_{median} and after evolution MLP - GP.

	Fixed Key		Random Keys	
	MLP - RS_{median}	MLP - GP	MLP - RS_{median}	MLP - GP
$\overline{Q}_{t_{GE}}$	2 377	1 168	3 350	$> 5\,000$

TABLE 7. Results of the EA effectiveness experiment for the ASCAD fixed and random keys datasets with the ID leakage model. The median architecture is compared before MLP - RS_{median} and after evolution MLP - GP.

	Fixed Key		Random Keys	
	MLP - RS_{median}	MLP - GP	MLP - RS_{median}	MLP - GP
$\overline{Q}_{t_{GE}}$	531	279	437	188

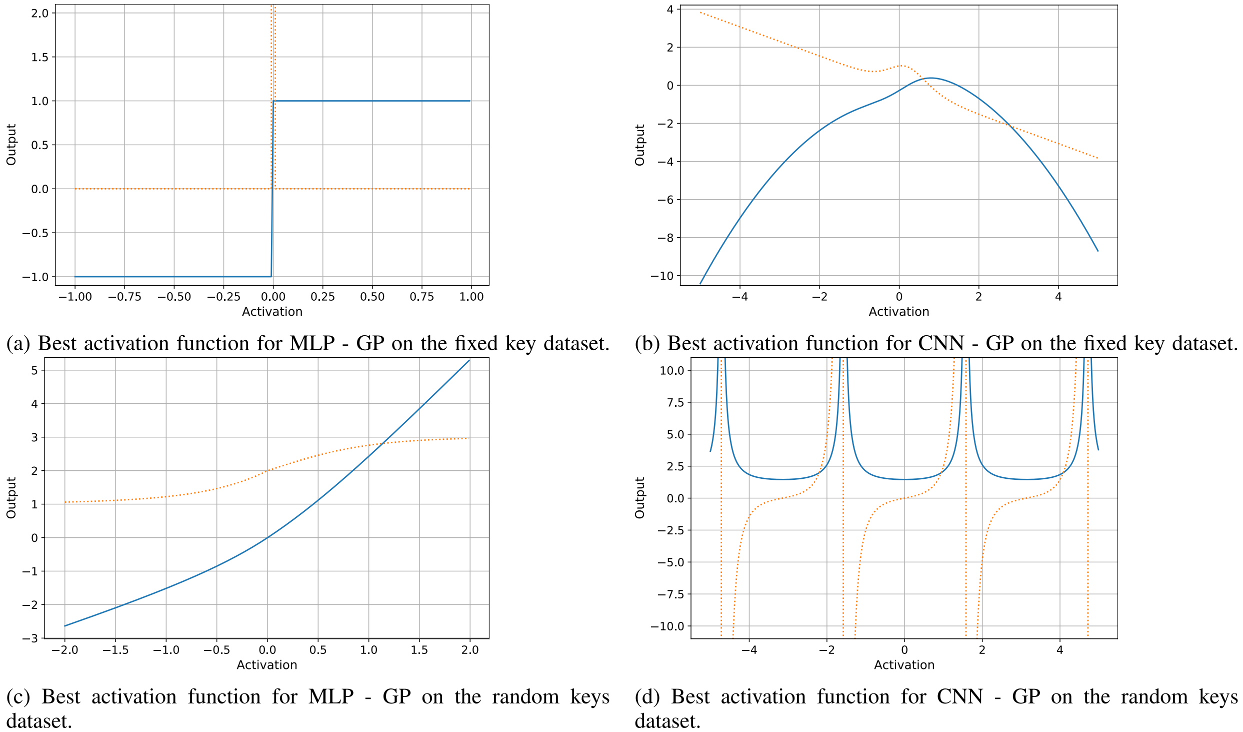
Interestingly, we can see that using evolved activation function allows that even median models perform not much worse than state-of-the-art results.

VII. DISCUSSION

To better understand the evolutionary process and the evolved activation functions, we perform a more detailed analysis of the obtained results for some selected experiments. We especially focus on the evolved activation functions to examine how they compare to existing activation functions.

A. CONVERGENCE ANALYSIS

In Figure 2, we show the GP evolution convergence diagrams for the MLP and CNN architectures for the ID leakage model on both ASCAD dataset versions. For the fixed key dataset, one can see a fairly fast convergence towards better solutions at the beginning of the evolution process, which then gradually slows down. After about 60 iterations, no further improvements can be observed. This result shows that GP had enough time to converge but probably converged to a local optimum. Therefore, it would be useful to introduce more diversity into the population in later iterations, either in the form of greater mutation probability or by replacing parts of the population with new, randomly generated individuals. Particularly strong convergence is observed for CNN, where the final fitness is more than twice as high as in the case of MLP. On the other hand, for the random keys dataset, we see that during the entire evolutionary process, the fitness stayed mostly the same, which shows that GP was unable to obtain any satisfactory activation functions. Similar convergence patterns were also observed for the HW leakage model. Therefore, we do not consider them further. These patterns suggest that the evolutionary process could have been terminated sooner to improve the execution time of the procedure.



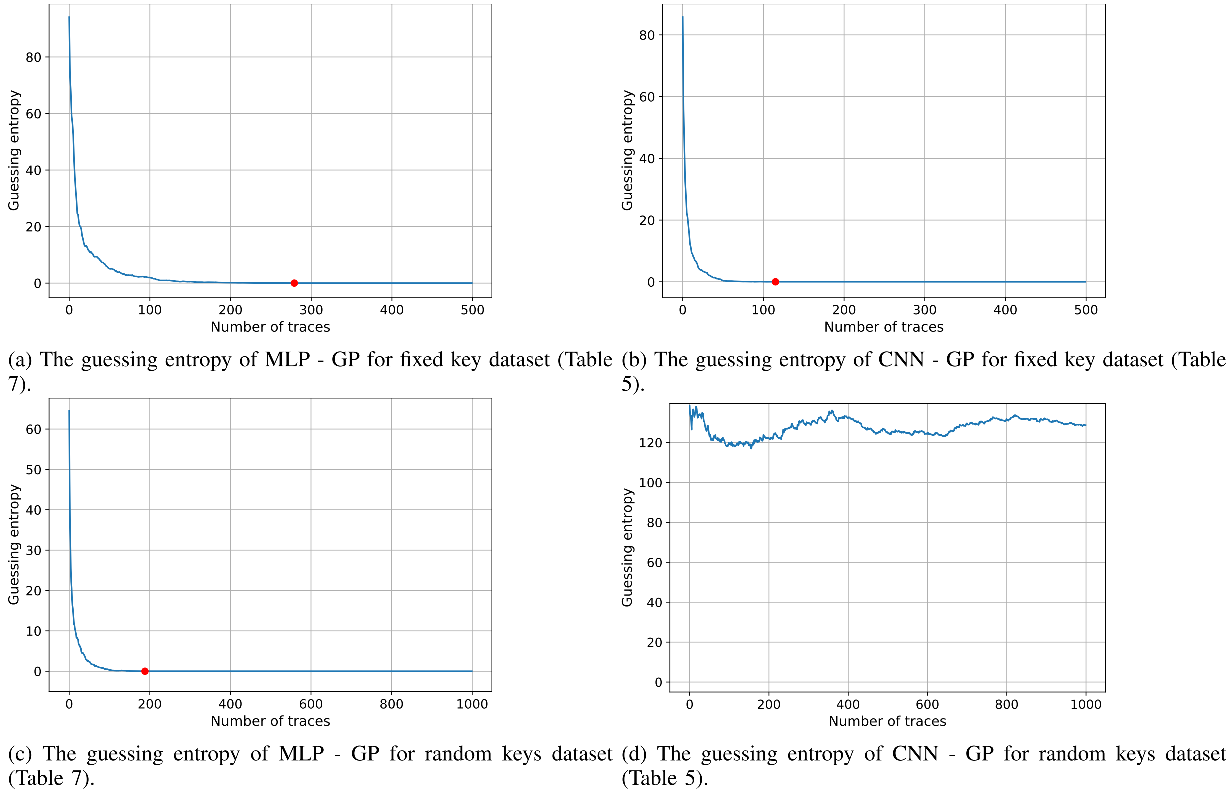
(a) Best activation function for MLP - GP on the fixed key dataset.

(b) Best activation function for CNN - GP on the fixed key dataset.

(c) Best activation function for MLP - GP on the random keys dataset.

(d) Best activation function for CNN - GP on the random keys dataset.

FIGURE 3. Plots represent the 1D slice of the best activation function (solid) and its derivative (dotted) obtained through evolution on the ASCAD dataset for the ID leakage model. The slice is defined by setting the first dimension to the x-axis and the second dimension is set to 0.



(a) The guessing entropy of MLP - GP for fixed key dataset (Table 7).

(b) The guessing entropy of CNN - GP for fixed key dataset (Table 5).

(c) The guessing entropy of MLP - GP for random keys dataset (Table 7).

(d) The guessing entropy of CNN - GP for random keys dataset (Table 5).

FIGURE 4. Guessing entropy of the best-evolved activation functions on the ASCAD dataset and the ID leakage model.

B. EVOLVED ACTIVATION FUNCTIONS

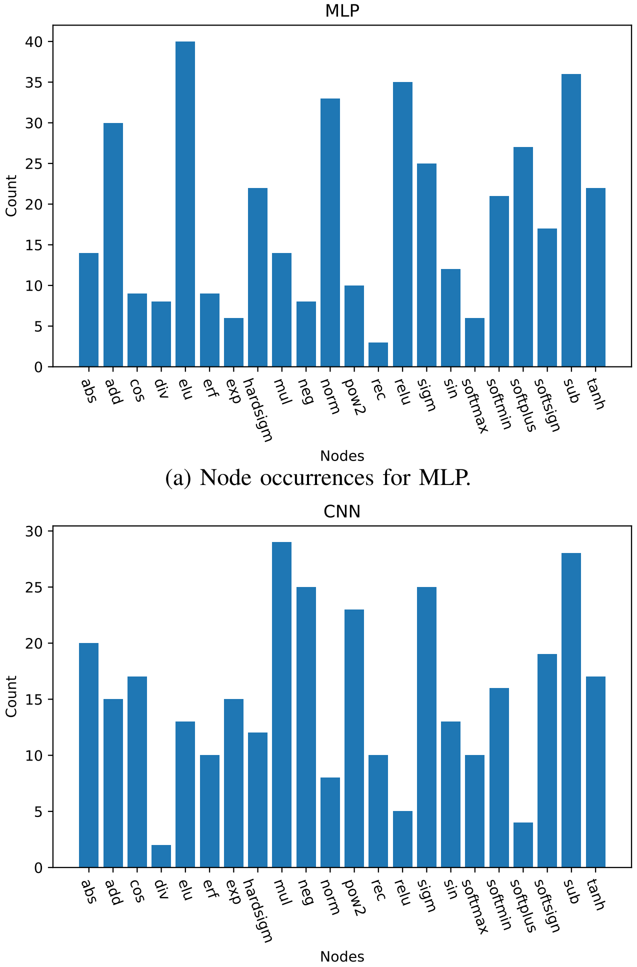
We selected some of the best-evolved functions for further analysis. We consider only the ID leakage model since the observed behavior is similar to the HW leakage model. The four best activation functions are listed below, where the first index in the subscript represents the ASCAD dataset (fixed key or random keys) and the second index represents the architecture type (CNN or MLP). Note that the functions look quite complex, and it is unlikely that a human designer would develop them. All of the activation functions, except $a_{RN,C}$, are composed of multiple functions put together. This suggests that the simple activation functions are not as powerful individually as they are when applied in conjunction. The experimental results show that they perform very well on the problem under consideration regardless of their complexity.

$$\begin{aligned} a_{FK,C}(\vec{x}) &= sin(erf(\vec{x} - softmin(\vec{x})^2)) \\ &\quad -\vec{x}^2 \cdot softsign(\sigma(softmin(softsign(\sigma_H(\sigma \\ &\quad \times(normalized(softmax(\vec{x}^2)))))))) \quad (4) \\ a_{FK,M}(\vec{x}) &= normalized(softsign(normalized(\vec{x} \\ &\quad \cdot tanh(softplus(softsign(erf(\vec{x}))))))) \quad (5) \\ a_{RN,C}(\vec{x}) &= (tanh(|sin(cos(\vec{x}))|))^{-1} \quad (6) \\ a_{RN,M}(\vec{x}) &= softplus(\vec{x} + ELU(\vec{x})) + \vec{x} \\ &\quad -\sigma(ELU(cos(\sigma_H(cos(softplus(\sigma(\vec{x}))))))) \quad (7) \end{aligned}$$

To get a better insight into the best function, we plot the best activation function and its derivative in Figure 3. Recall that the derivative is important as we need a differentiable function when we use the backpropagation algorithm common in neural network training. For the fixed key dataset and MLP, the best-obtained activation function looks quite similar to a step function, whereas, for CNN, we obtain a non-monotonic function that is strikingly different from the activation functions commonly used (as well as from state-of-the-art). For the random keys dataset, the functions obtained for MLP resemble those commonly used, while the form of the function for CNN is rather unusual (it is periodic, which is not a common case with activation functions).

In Figure 4 we show the number of attack traces required to achieve a guessing entropy of 0. This value is marked with a red dot. For the fixed key dataset, we reach the target much faster using CNN than MLP, but both techniques perform well. This ensures that we can find custom activation functions for SCA that work well regardless of neural network choice. On the other hand, for the random keys dataset, MLP performs very well, while CNN does not converge. Again, this shows that it is easier to optimize MLP architectures, whether via hyperparameter tuning [30] or activation function evolution, as studied here. Moreover, the results suggest that MLP architectures are sufficient to break the targets, especially when there is no trace misalignment (we only consider synchronized traces in this work). For CNNs, working with larger mutation rates and longer evolution could solve the problems mentioned here.

To get a better insight into which function nodes are most useful for GP, we summarise the occurrence of these nodes for the three best individuals for all results. Figure 5 shows the histograms of the occurrence of function nodes for the MLP and CNN models. The figure shows that different function nodes are preferred for each model, such as *sub*, *sigm*, and *tanh*. Similarly, certain activation functions, such as *div*, *rec*, and *softmax*, are rarely used by the models. However, the situation for the other function nodes is not as clear since the number of their occurrences depends largely on the model that is used. The unfortunate consequence of this is that it is difficult to reduce the number of function nodes used in the evolution process because, with some exceptions, it is difficult to determine which functions generally perform poorly.



(a) Node occurrences for MLP.

(b) Node occurrences for CNN.

FIGURE 5. Function node occurrences in the evolved activation functions.

VIII. CONCLUSION AND FUTURE WORK

This paper investigates how neuroevolution can improve deep learning-based side-channel analysis. More specifically, we consider the setting where genetic programming evolves activation functions specifically adapted for side-channel analysis. We conduct experiments for two SCA datasets and two leakage models to show that it is possible to evolve activation functions that improve attack behavior. We find that the evolution of activation functions is more efficient for a simpler dataset, suggesting that more work is needed to understand the advantages and disadvantages of this approach. We also find that more informative and cost-effective fitness functions are needed, which would lead to better individuals more quickly.

As this is one of the first works to consider neuroevolution for SCA, there are many possible research directions for future work. One direction would be to consider evolving other elements of the learning process, such as the loss function. It would also be interesting to investigate how transferable the obtained activation functions are when considering already evolved neural network architectures. Therefore, in future studies, we aim to examine the evolved activation functions with several different neural network architectures to investigate this issue. Finally, a common problem associated with neural networks and activation functions is gradient vanishing. Although this problem was not considered in the current study, it represents an important follow up direction which should be examined to determine whether the evolved functions are susceptible to it, and if yes, how the problem could be mitigated.

REFERENCES

[1] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Cham, Switzerland: Springer, Dec. 2006. [Online]. Available: http://www.springer.com/

[2] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. 19th Annu. Int. Cryptol. Conf. Adv. Cryptol.* London, U.K.: Springer-Verlag, 1999, pp. 388–397.

[3] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards," in *Smart Card Programming and Security*, I. Attali and T. Jensen, Eds. Berlin, Germany: Springer, 2001, pp. 200–210.

[4] P. C. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Proc. CRYPTO* in Lecture Notes in Computer Science, vol. 1109. Cham, Switzerland: Springer, 1996, pp. 104–113.

[5] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), vol. 2523. Cham, Switzerland: Springer, Aug. 2002, pp. 13–28.

[6] 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 Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, pp. 148–179, May 2019.

[7] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, "Methodology for efficient CNN architectures in profiling attacks," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 1–36, Nov. 2019.

[8] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 1997–2017, Jan. 2019.

[9] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, Aug. 2017, pp. 1126–1135.

[10] S. Gonzalez and R. Miikkulainen, "Improved training speed, accuracy, and data utilization through loss function optimization," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.

[11] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search (version 2)," 2019, *arXiv:1905.01392*, doi: 10.48550/ARXIV.1905.01392.

[12] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 1294–1303.

[13] M. Feurer, J. T. Springenberg, and F. Hutter, "Initializing Bayesian hyperparameter optimization via meta-learning," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 1128–1135.

[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016, *arXiv:1611.01578*, doi: 10.48550/ARXIV.1611.01578.

[15] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, "Language modeling with gated convolutional networks," 2016, *arXiv:1612.08083*.

[16] A. L. Maas, "Rectifier nonlinearities improve neural network acoustic models," 2013.

[17] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Int. Conf. Mach. Learn.*, Madison, WI, USA: Omnipress, 2010, pp. 807–814.

[18] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," 2018, *arXiv:1811.03378*, doi: 10.48550/ARXIV.1811.03378.

[19] 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*, A. Joux, Ed. Berlin, Germany: Springer, 2009, pp. 443–461.

[20] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997.

[21] R. Collobert and S. Bengio, "Links between perceptrons, MLPs and SVMs," in *Proc. 21st Int. Conf. Mach. Learn. (ICML)*, 2004, p. 23.

[22] Y. Lecun and Y. Bengio, *Convolutional Networks for Images, Speech, and Time-Series*. Cambridge, MA, USA: MIT Press, 1995.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[24] W. Duch and N. Jankowski, "Survey of neural transfer functions," *Neural Comput. Surv.*, vol. 2, no. 1, pp. 163–213, 1999.

[25] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[26] H. Maghrebi, T. Portigliatti, and E. Prouff, "Breaking cryptographic implementations using deep learning techniques," in *Proc. Int. Conf. Secur., Privacy, Appl. Cryptogr. Eng.* Cham, Switzerland: Springer, 2016, pp. 3–26.

[27] E. Cagli, C. Dumas, and E. Prouff, "Convolutional neural networks with data augmentation against jitter-based countermeasures," in *Cryptographic Hardware and Embedded Systems CHES 2017*, W. Fischer and N. Homma, Eds. Cham, Switzerland: Springer, 2017, pp. 45–68.

[28] G. Perin, L. Chmielewski, and S. Picek, "Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 337–364, Aug. 2020.

[29] J. Rijsdijk, L. Wu, G. Perin, and S. Picek, "Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 3, pp. 677–707, Jul. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8989

[30] L. Wu, G. Perin, and S. Picek, "I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis," *IEEE Trans. Emerg. Topics Comput.*, early access, Nov. 7, 2022, doi: 10.1109/TETC.2022.3218372.

[31] L. Wouters, V. Arribas, B. Gierlichs, and B. Preneel, "Revisiting a methodology for efficient CNN architectures in profiling attacks," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 147–168, Jun. 2020.

[32] R. Y. Acharya, F. Ganji, and D. Forte, "InfoNEAT: Information theory-based NeuroEvolution of augmenting topologies for side-channel analysis," 2021, *arXiv:2105.00117*.

[33] 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 Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, pp. 235–274, Jul. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8974

[34] G. Perin, L. Wu, and S. Picek, "Exploring feature selection scenarios for deep learning-based side-channel analysis," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 828–861, Aug. 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9842

[35] C. Pfeifer and P. Haddad, "Spread: A new layer for profiled deep-learning side-channel attacks," Cryptol. ePrint Arch., Paper 2018/880, 2018. [Online]. Available: https://eprint.iacr.org/2018/880

[36] G. Zaid, L. Bossuet, F. Dassance, A. Habrard, and A. Venelli, "Ranking loss: Maximizing the success rate in deep learning side-channel analysis," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, pp. 25–55, Dec. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8726

[37] J. Zhang, M. Zheng, J. Nan, H. Hu, and N. Yu, "A novel evaluation metric for deep learning-based side channel analysis and its extended application to imbalanced data," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 73–96, Jun. 2020.

[38] M. Kerkhof, L. Wu, G. Perin, and S. Picek, "Focus is key to success: A focal loss function for deep learning-based side-channel analysis," in *Constructive Side-Channel Analysis and Secure Design*. Cham, Switzerland: Springer, 2022, pp. 29–48, doi: 10.1007/978-3-030-99766-3_2.

[39] M. Kerkhof, L. Wu, G. Perin, and S. Picek, "No (good) loss no gain: Systematic evaluation of loss functions in deep learning-based side-channel analysis," Cryptol. ePrint Arch., Paper 2021/1091, 2021. [Online]. Available: https://eprint.iacr.org/2021/1091

[40] S. Koziel, P. Mahouti, N. Calik, M. A. Belen, and S. Szczepanski, "Improved modeling of microwave structures using performance-driven fully-connected regression surrogate," *IEEE Access*, vol. 9, pp. 71470–71481, 2021.

[41] K. Lee and J. Yim, "Hyperparameter optimization with neural network pruning," 2022, *arXiv:2205.08695*.

[42] Y. Liu and X. Yao, "Evolutionary design of artificial neural networks with different nodes," in *Proc. IEEE Int. Conf. Evol. Comput.*, May 1996, pp. 670–675.

[43] A. Marchisio, M. A. Hanif, S. Rehman, M. Martina, and M. Shafique, "A methodology for automatic selection of activation functions to design hybrid deep neural networks," 2018, *arXiv:1811.03980*, doi: 10.48550/ARXIV.1811.03980.

[44] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, 2002.

[45] A. Hagg, M. Mensing, and A. Asteroth, "Evolving parsimonious networks by mixing activation functions," in *Proc. Genetic Evol. Comput. Conf.*, Jul. 2017, pp. 425–432.

[46] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017, *arXiv:1710.05941*, doi: 10.48550/ARXIV.1710.05941.

[47] G. Bingham, W. Macke, and R. Miikkulainen, "Evolutionary optimization of deep learning activation functions," in *Proc. Genetic Evol. Comput. Conf.*, Jun. 2020, pp. 289–296.

[48] M. Basirat and P. M. Roth, "The quest for the golden activation function," 2018, *arXiv:1808.00783*.

[49] G. Bingham, W. Macke, and R. Miikkulainen, "Evolutionary optimization of deep learning activation functions," in *Proc. Genetic Evol. Comput. Conf.*, Jun. 2020, pp. 289–296, doi: 10.1145/3377930.3389841.

[50] A. Nader and D. Azar, "Evolution of activation functions: An empirical investigation," 2021, *arXiv:2105.14614*.

[51] K. Vijayaprabakaran and K. Sathiyamurthy, "Neuroevolution based hierarchical activation function for long short-term model network," *J. Ambient Intell. Humanized Comput.*, vol. 12, no. 12, pp. 10757–10768, Jan. 2021, doi: 10.1007/s12652-020-02889-w.

[52] R. Lapid and M. Sipper, "Evolution of activation functions for deep learning-based image classification," in *Proc. Genetic Evol. Comput. Conf. Companion*, Jul. 2022, pp. 2113–2121, doi: 10.1145/3520304.3533949.

[53] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ASCAD database," *J. Cryptograph. Eng.*, vol. 10, no. 2, pp. 163–188, 2020.

[54] 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 Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, pp. 209–237, Nov. 2018.

[55] F.-A. Fortin, F.-M. D. Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Lang. Res.*, vol. 13, pp. 2171–2175, Jul. 2012.

[56] A. Paszke, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2019, pp. 8024–8035.

KARLO KNEŽEVIĆ (Member, IEEE) received the B.Sc. and M.Sc. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2011 and 2013, respectively. He is currently pursuing the Ph.D. degree with the Faculty of Electrical Engineering and Computing, University of Zagreb. He is currently a Team Lead of analytics and data science with SofaScore Company, and an Associate Lecturer with Algebra University College. His research interests include evolutionary computation, machine learning, and symmetric cryptography.

JURAJ FULIR received the B.Sc. and M.Sc. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2017 and 2019, respectively. He is currently pursuing the Ph.D. degree in fusing geoinformation in raster and vector form for the procedural generation of interactive virtual environments. He is currently a Younger Researcher with the Faculty of Electrical Engineering and Computing, University of Zagreb.

DOMAGOJ JAKOBOVIĆ (Senior Member, IEEE) received the B.Sc. degree in December 1996, the M.Sc. degree in electrical engineering, in December 2001, and the Ph.D. degree in generating scheduling heuristics with genetic programming, in December 2005. He is currently a Full Professor with the Faculty of Electrical Engineering and Computing, University of Zagreb. Since April 1997, he is a member of the Research and Teaching Staff with the Department of Electronics, Microelectronics, Computer and Intelligent Systems of the Faculty of Electrical Engineering and Computing, University of Zagreb.

STJEPAN PICEK (Senior Member, IEEE) received the Ph.D. degree in 2015. From 2015 to 2017, he was a Postdoctoral Researcher at KU Leuven, Belgium, and MIT, USA. From 2017 to 2021, he was an Assistant Professor at the Delft University of Technology, The Netherlands. He is currently an Associate Professor with Radboud University, The Netherlands. His research interests include security, machine learning, and evolutionary algorithms.

MARKO ĐURASEVIĆ (Member, IEEE) received the B.Sc. and M.Sc. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2012 and 2014, respectively, and the Ph.D. degree in generating dispatching rules for solving scheduling problems in an unrelated machine environment, in February 2018. He is currently an Assistant Professor with the Faculty of Electrical Engineering and Computing, University of Zagreb.