NeuroSCA: Evolving Activation Functions for Side-channel Analysis

Karlo Knezevic[1], Juraj Fulir[1], Domagoj Jakobovic[1] and Stjepan Picek[2]

[1] University of Zagreb, Croatia
[2] Delft University of Technology, The Netherlands

Abstract. The choice of activation functions can have a significant effect on the performance of a neural network. Although the researchers have been developing novel activation functions, Rectified Linear Unit ($ReLU$) remains the most common one in practice. This paper shows that evolutionary algorithms can discover new activation functions for side-channel analysis (SCA) that outperform $ReLU$. Using Genetic Programming (GP), candidate activation functions are defined and explored (neuroevolution). As far as we know, this is the first attempt to develop custom activation functions for SCA. The ASCAD database experiments show this approach is highly effective compared to the state-of-the-art neural network architectures. While the optimal performance is achieved when activation functions are evolved for the particular task, we also observe that these activation functions show the property of generalization and high performance for different SCA scenarios.

Keywords: Activation functions · Multilayer perceptron · Convolutional neural network · Side-channel analysis · Evolutionary Algorithms · Neuroevolution

1 Introduction

Modern digital systems are commonly equipped with cryptographic primitives, acting as the foundation of security, trust, and privacy protocols. While such primitives are proven to be mathematically secure, poor implementation choices can make them vulnerable to attackers. Such vulnerabilities are commonly known as leakage [MOP06]. Side-channel leakage exploits various sources of information leakage in the device where some common examples of leakage are timing [Koc96], power [KJJ99b], and electromagnetic (EM) emanation [QS01] and the attacker is a passive one. The researchers proposed several side-channel analysis (SCA) approaches to exploit those leakages in the last few decades. One common division of side-channel analyses is into non-profiling and profiling attacks. Non-profiling attacks like Simple Power Analysis (SPA) [KJJ99a] or Differential Power Analysis (DPA) [KJJR11] require fewer assumptions but could need thousands of measurements (traces) to break a target, especially if it is protected with countermeasures. On the other hand, profiling attacks are considered as one of the strongest possible attacks [CRR02]. There, the attacker has full control over a clone device, which he uses to build device's complete profile. The attacker then uses this profile to target other similar devices to recover the secret information. The deep learning approaches represent a powerful (and more recent) option for profiling SCA. Indeed, the results in the last few years show the potential of such an approach where neural networks like multilayer perceptron (MLP and convolutional neural networks (CNNs) can break targets protected with countermeasures [KPH+19, ZBHV19]. Still, finding high-performing neural network architectures is often not an easy task due to a large number of hyperparameters to consider. In the hyperparameter tuning phase, we can distinguish between two rather different approaches. The first approach considers various

techniques to select the best-performing hyperparameters [EMH19, FSH15, FAL17, FL18, GM20, WRP19]. Common techniques include gradient descent, Bayesian hyperparameter optimization, reinforcement learning, and evolutionary algorithms [CXWT19, FSH15, ZL17]. The second direction considers the design of neural network architectures or neural network elements like loss functions or activation functions. While it is possible to design various activation functions, only a small number of activation functions are widely used in modern neural network architectures. For instance, the Rectified Linear Unit, $ReLU(x) = max\{x, 0\}$, is popular due to its simplicity and effectiveness [BMM20]. There are also attempts to design new activation functions that have certain properties, but none achieved widespread adoption like $ReLU$.

This paper develops an evolutionary approach to evolve activation functions for side-channel analysis. We build upon recent results considering deep learning architectures. We ask whether it is possible to make deep learning-based SCA even more efficient if activation functions in a neural network are optimized for a particular problem (neural network architecture, side-channel leakage model, and dataset). More precisely, we use Genetic Programming (GP), where we represent activation functions as syntactic trees, and we evolve custom expressions. The resulting functions are unlikely to be discovered manually, yet they perform (surprisingly) well, surpassing traditional activation functions like $ReLU$ on common side-channel measurements, like those in the ASCAD database. To the best of our knowledge, this is the first time that neuroevolution is used for SCA or that evolutionary algorithms are used to develop activation functions for SCA.

The two main contributions of this paper are:

1. We evolve novel activation functions used in multilayer perceptron and Convolutional Neural Network. Neural networks with those activation functions show better performance comparing to existing relevant research. **This shows that the newly developed activation functions have their place in the future designs of neural network architectures for SCA.**
2. We replace popular activation functions with evolved activation functions in previously developed neural network topologies and demonstrate better network performance after this substitution. **This shows that optimization of the activation function has relevance even when considering already developed neural networks.**

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

The remainder of this paper is organized as follows. Section 2 covers the necessary definitions and notions. Section 3 discusses related works. Section 4 describes the datasets and parameters that we considered. Section 5 presents the results obtained by our experiments. Finally, Section 6 sums up the key contributions of the paper and gives possible avenues for future work.

2 Background

2.1 Notation

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

We define a dataset as a collection of traces (measurements) $\mathbf{D}$. Each trace $\mathbf{x}_i$ is 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. We divide the dataset into three parts: profiling set consisting of N traces, validation set consisting of V traces, and attack set consisting of Q traces.

We denote the vector of learnable parameters in our profiling models as $\boldsymbol{\theta}$ and the set of hyperparameters defining the profiling model as $\mathcal{H}$. We consider the supervised machine learning task (classification), where the goal is to predict the class value $v \in V$ for an input x. The size of the set V equals c.

2.2 Machine Learning-based SCA

We consider a typical profiling side-channel analysis setting with two phases: training (profiling) and testing (attack). A powerful attacker has a device (clone device) with knowledge about the secret key. The attacker can obtain a set of N profiling traces $x_1, \ldots, x_N$ (where each trace corresponds to the processing of plaintext or ciphertext i).

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

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

where k^* represents the secret (unknown) key on the device under the attack.

We consider an attack on a block cipher (the AES cipher) and conduct 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 examples of input-output pairs, where the number of classes c is determined by the leakage model. The function f is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^n$, where n denotes the number of trainable parameters.

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 $Q \times c$. Every element $\mathbf{p}_{i,v}$ of matrix P is a vector of all class probabilities for a specific trace $\mathbf{x}_i$. The probability $S(k)$ for any key byte candidate k is used as an log-likelihood distinguisher:

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

The value $\mathbf{p}_{i,v}$ denotes the probability that for a key k and input i_i, the result is class v (derived from the key and input through a cryptographic function and a leakage model l).

Finally, to estimate the effort required to break the secret key, it is common to use the guessing entropy (GE) [SMY09] metric. An attack outputs 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. Here, g_1 is the most likely key candidate and $g_{|\mathcal{K}|}$ the least likely key candidate. Guessing entropy is the average position of k^* in $\mathbf{g}$. Our work considers attacks on specific key bytes only, formally using the partial guessing entropy metric. Still, due to simplicity, we denote it as guessing entropy.

2.3 Artificial Neural Network

Artificial neural networks (ANNs) is a notion for all computer systems loosely inspired by biological neural networks. Such systems can "learn" from examples, making them a very popular paradigm in the machine learning domain. Any ANN is built from a number of nodes called artificial neurons, where the nodes are connected to transmit a signal.

A very simple type of neural network is called a perceptron. A perceptron is a linear binary classifier applied to the feature vector as a function that decides whether or not an input belongs to some categorical class. Each vector component has an associated weight w_i, and each perceptron has a threshold value q. A perceptron's output equals "1" if the direct sum between the feature vector and the weight vector is larger than the threshold

value and "-1" otherwise. A perceptron classifier works only for linearly separable data, i.e., when there is some hyperplane that separates all the positive points from all the negative points [Mit97].

2.3.1 Multilayer Perceptron

By integrating more layers to a perceptron, we obtain a multilayer perceptron algorithm. Multilayer perceptron (MLP) is a feed-forward neural network that maps sets of inputs onto sets of appropriate outputs. Differing from linear perceptron, MLP can distinguish data that are not linearly separable. MLP consists of multiple layers of nodes in a directed graph, where each layer is connected to the next one. Consequently, each node in one layer connects with a specific weight w to every node in the following layer. Multilayer perceptron algorithm consists of at least three layers: one input layer, one output layer, and one hidden layer. Those layers must consist of nonlinearly activating nodes [CB04]. The backpropagation algorithm is utilized for training the network, which is a generalization of the least mean squares algorithm in the linear perceptron. The gradient descent optimization algorithm utilizes backpropagation to adjust the weight of neurons by calculating the gradient of the loss function [Mit97].

2.3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) represent a type of neural networks first designed for 2-dimensional convolutions as it was inspired by the biological processes of animals' visual cortex [Fuk80, LB95]. From the operational perspective, CNNs are similar to ordinary neural networks (e.g., multilayer perceptron). More precisely, they consist of a number of layers where each layer is composed of neurons. CNNs use three main types of layers: convolutional layers, pooling layers, and fully-connected layers. Convolutional layers are linear layers that share weights across space. Pooling layers are nonlinear layers that minimize the spatial size in order to inhibit the number of neurons. Fully-connected layers are layers where every neuron is connected with all the neurons in the neighborhood layer (as in the MLP). For additional information about CNNs, we refer interested readers to [GBC16].

2.3.3 Activation Functions

An activation function of a node is a function g defining the output of a node given an input or set of inputs from a layer of linear nodes, as denoted in Eq. (2). To enable replications of nontrivial functions with ANNs using 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 to the bias value allow the activation function to be shifted across the input domain, while changes to the weights alter the activation function's steepness. Combined with the backpropagation algorithm, this enables the ANN to model the data automatically.

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, which prohibits multi-classification. A linear activation function suffers from two major problems: it cannot use gradient descent to train the model because the derivative of the function is a constant, and all layers of the neural network collapse into one. Note, a linear combination of linear functions is still a linear function, and such activation function transforms the neural network into a single layer [GBC16].

As a result, modern neural network models use nonlinear activation functions. They allow the model to create complex mappings between the network's inputs and outputs, essential for learning and modeling complex data or data with high dimensionality. Nonlinear functions address the problems of linear activation functions. First, they allow backpropagation learning because they have a derivative function that is related to the inputs. What is more, 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 by empirical results and speed of execution. Although some functions have theoretically substantiated properties, that does not make them the best in practice. Typically, *ReLU* is often used, and for the hidden layers of recursive models, *tanh* is commonly selected, see, e.g., [DJ99].

2.4 Genetic Programming

Genetic programming (GP) is an automated optimization process for developing computer programs, used to solve complex problems in computing and the problems we encounter every day. The concept is based on general ideas derived from the theory of genetic algorithms and other evolutionary methods. Simply put, the ultimate goal of GP (as a product) is a universal computer program that finds solutions to problems described only with input data and the desired results.

Although genetic programming can be applied in many different ways and from many different perspectives, in most cases, it represents a computer program as a syntactic tree in the context of graph theory. Namely, each computer program can be represented as a tree or a forest of trees (in a broader sense), where the tree's internal nodes have the role of operators (or functions of a certain number of variables), and the leaves the role of operands. The set of operators (the function set) and operands (the terminal set) are predefined for a specific task. The evolution typically begins with a population consisting of randomly generated candidate solutions. These are called individuals, whose operators and operands can be altered by three bio-inspired operations: selection, crossover, and mutation.

Selection is a process whereby certain individuals are selected from the current generation according to their fitness, thus serving as parents for the next generation. The individuals are selected probabilistically such that the better performing individuals have a higher chance of getting selected. In contrast, the crossover is a binary operator, exchanging information between two individuals to form a new offspring. Similarly, the mutation is also a stochastic operator that helps increase the population's diversity by randomly choosing one or more nodes in an offspring and changing them.

Then, in an iterative process, where an iteration is referred to as generation, each new individual's fitness is evaluated. Based on their fitness, we select parent solutions for breeding. Subsequently, we apply breeding operators on pairs of individuals to generate new pairs of offsprings. The process is repeated until a predefined number of generations or another stopping criterion is met.

2.4.1 Activation Function as a Tree

In our work, each activation function is represented as a tree consisting of unary and binary operators. The terminals in the tree represent the function input values, while the root node's value represents the output function value. Each individual in the GP population is a potential candidate activation function. An individual is evaluated by using the activation function it embodies in a neural network. The individual's fitness is then defined as the neural network's performance applied to a specific task - in this case, the key prediction efficiency.

3 Related Work

Finding better (custom) activation functions for SCA is one perspective of hyperparameter tuning. Up to now, in SCA, most of the works considered hyperparameter tuning but using common options (i.e., not designing new neural network elements). There, we can enumerate several phases in profiling SCA and hyperparameter tuning. The first approaches in profiling SCA like template attack [CRR02]), or machine learning-based attacks (random forest [LMBM13], support vector machines [HZ12, PHJ+17], Naive Bayes [PHG17]) had only a few or even none hyperparameters to tune.

In 2016, Maghrebi et al. introduced convolutional neural networks for profiling SCA [MPP16]. The authors also reported they used genetic algorithms to tune the hyperparameters. While it is difficult to know whether this is the first time deep learning was used in SCA (many works omitted details about neural network architectures), this work represented a significant turning point in the SCA research. Indeed, from this moment, the SCA community moved its attention from other profiling methods to (almost exclusively) deep learning.

As a result, there are multiple research works reporting very good attack performance even in the presence of countermeasures [CDP17, PHJ+18]. Interestingly, the first work did not discuss hyperparameter tuning, while the second one conducted manual hyperparameter tuning. Kim et al. constructed VGG-like architecture that performs well over several datasets, but they did not discuss the hyperparameter tuning involved in checking the performance of such an architecture [KPH+19]. Benadjila et al. made an empirical evaluation of different CNN hyperparameters for the ASCAD dataset [BPS+20]. Perin et al. used a random search in predefined ranges to build deep learning models to form ensembles [PCP20]. Both of those works reported very good results, despite relatively simple methods to choose hyperparameters. Wu et al. proposed to use Bayesian optimization to find optimal hyperparameters for MLP and CNN architectures [WPP20]. Their results indicated it is possible to find excellent architectures and that even random search can find many architectures that exhibit top performance. Rijsdijk et al. explored how reinforcement learning can be used for hyperparameter tuning for CNNs [RWPP21]. They reported very good results (attack performance with small neural network architectures), but their approach requires significant computational resources.

Besides improving the neural network performance by conducting efficient hyperparameter tuning, several works aim to provide a methodology to build neural networks for SCA. Zaid et al. proposed a method to select hyperparameters related to the size (number of learnable parameters, i.e., weights and biases) of layers in CNNs. The authors considered the number of filters, kernel sizes, strides, and the number of neurons in fully-connected layers [ZBHV19]. Wouters et al. [WAGP20] improved upon the work from Zaid et al. [ZBHV19], and discussed several problems in the original work. More precisely, Wouters et al. showed how to reach similar attack performance with significantly smaller neural network architectures.

Some works also concentrate on improving the performance of deep learning-based SCA by designing custom neural network elements. Pfeifer and Haddad designed a new type of layer called “Spread”, and they claimed it reduces the number of layers required and speeds up the learning phase [PH18]. Zheng et al. proposed a new metric function called Cross Entropy Ratio (CER), where they adapted it to a new loss function, designed specifically for deep learning in SCA [ZZN+20]. Zaid et al. introduced a new loss function derived from the learning to rank approach that helps to prevent approximation and estimation errors [ZBD+20].

On the other hand, several works investigate the evolution of activation functions by using evolutionary algorithms and machine learning. A first attempt to learn activations in a neural network can be found in [LY96], where the authors proposed to randomly add or remove logistic or Gaussian activation functions using genetic programming. In [MHR+18],

the authors developed a method to automatically select an activation function for each layer of a neural network. More precisely, they identify the evaluation points during the learning process to evaluate the accuracy and perform early stopping if needed. Hagg et al. improved over the NEAT algorithm [SM02] to concurrently evolve network topology, weights, and activation functions of neurons [HMA17]. Ramachandran et al. investigated the combination of reinforcement learning and exhaustive search to design activation functions automatically [RZL17]. Their experiments resulted in many good activation functions, but their analysis concentrated on one activation function ($x \cdot \sigma(x)$) that they report works better than $ReLU$ on deeper neural network models. Bingham et al. augmented previous research by introducing an evolutionary algorithm to design novel activation functions [BMM20]. The authors showed that it is possible to evolve specialized activation functions that perform well for the CIFAR-10 and CIFAR-100 datasets. In [BR18], the authors use a hybrid genetic algorithm to evolve a function defined differently on the positive and negative domains. Parts of the function are represented by trees and crossed by special operators that separately change the positive and negative sides. The set of nodes comprises of basic arithmetic operations, and leaves are popular activation functions without constants. The authors also presented the new activation functions $ELiSH$ and Hard $ELiSH$, which they built manually, intending to combine smaller functions' good properties. On three datasets, they showed that their functions perform the best.

Finally, we briefly discuss works investigating activation functions designed manually. Nair and Hinton introduced rectified linear unit ($ReLU$), which is today de-facto standard activation function for deep learning [NH10]. Various $ReLU$ modifications were proposed over the years, where one example is leaky $ReLU$ ($LReLU$), which deals with the dead neurons issue [MHN13]. More variations of $ReLU$ can be found in [KMK15, HZRS15]. Clevert et al. experimented with exponential linear unit function (ELU), which reduces the vanishing gradient problem [CUH16]. Furthermore, Klambauer et al. extended properties of ELU with scaled exponential linear unit function ($SELU$) [KUMH17]. The authors in [KH19, MR18] proposed to use a combination of different activation functions in the same layer. However, this approach has memory issues, which is typically a critical parameter in real-world scenarios.

4 Experimental Setup

In this section, we first discuss the datasets and leakage models we consider. Afterward, we give details about the investigated approaches to design activation functions.

4.1 Datasets and Leakage Models

In our experiments, we consider two versions of the ASCAD database [BPS+20]. This database contains the measurements from an 8-bit AVR microcontroller running a masked AES-128 implementation. This database is publicly available from https://github.com/ANSSI-FR/ASCAD.

The first version of the ASCAD database has a fixed key and consists of 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 use 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, and the dataset consists of 200 000 traces for training and 100 000 for testing. 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 to a Gaussian distribution (zero mean and unit variance) for both datasets by calculating the training set's distribution parameters.

This paper considers the Hamming Weight (HW) leakage model and the Identity (ID) leakage model. In the HW leakage model, the attacker assumes the leakage is proportional to the sensitive variable's Hamming weight. When considering a cipher that uses an 8-bit S-box, this leakage model results in nine classes. Since this induces a heavy imbalance in the label distribution, we additionally calculate the imbalance weights that balance the calculated model loss. For calculating the imbalance weights, we follow the guidelines given in [PHJ+18]. For the ID leakage model, the attacker considers the leakage in the form of 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).

4.2 Architecture Search Strategies

In our experiments, we consider two neural network types: CNN and MLP. The specific CNN architecture is described in [ZBHV19] for the ASCAD synchronized dataset, and we use it with all of its reported hyperparameters. The procedure's seed values were not defined, so we translated their original architecture and training procedure from Keras to PyTorch and ran the process with several seed values until we observed an equal or better result than reported in the original paper. The exact hyperparameters are reported in Section 5. We used the same seed value throughout our experiments for this architecture. The authors implemented a one-cycle learning rate schedule, which we replaced with the implementation in PyTorch, following their hyperparameter setup. The network architecture consists of a convolutional layer with four output channels, followed by batch normalization, activation function, and average pooling. The output of this block is flattened and fed to an MLP tail to produce the final prediction. The width and depth of the MLP tail were optimized per dataset with the grid search. In the CNN training procedures, we use the one-cycle policy for learning rate with reported hyperparameters of learning rate 0.005, with 40% of the cycle incrementing the value using a linear annealing strategy.

Since CNN inference time can be quite long compared to MLP, we apply and compare both architectures. The MLP architecture is defined with consecutive blocks consisting of a dense linear layer, batch normalization layer, and a nonlinear activation function.

We first employ architecture search to find the representative architecture for each of the datasets (and neural network types). Two different algorithms were used to explore the space of network architectures and their hyperparameters: grid search and random search. Both techniques include evaluating a multitude of points of search space to find the optimal one. The points are evaluated on the test set to obtain a distribution of representative solutions that will later serve for further optimization with evolution. The techniques are also easily parallelized, allowing a much faster search process than sequential evaluation. We consider grid search for CNNs as the number of hyperparameters is large, making it more difficult for a random search. For MLP, we use random search as related works reported good results even with such a simple tuning setup [PCP20, WPP20].

4.2.1 Grid Architecture Search for CNNs

Grid search evaluates all possible combinations of parameter values for a given search space, where the continuous variables are sampled along with fixed steps. We employ this search strategy following [ZBHV19] to find an optimal CNN architecture for a given dataset. Due to time constraints, we slightly truncated this search space by removing hyperparameter values that were not expected to give good results (such as very shallow or narrow architectures). The considered hyperparameter space is described in Table 1 and resulted in 2160 grid samples to obtain the best convolutional model (CNN - GS_{best}) for each of the datasets.

4.2.2 Random Architecture Search for MLP

Random search strategy samples the given space by randomly selecting points in the search space. The search space can be defined with a multitude of features, both discrete and continuous. It is similar to grid search but without a structured sampling of continuous spaces, which might introduce bias by selecting from only a subset of possible values. We find this bias unwanted as we do not use a learning rate schedule for our MLP models, and the model might be more sensitive to a fixed learning rate's exact value. We define the search space as the collection of hyperparameters that affect the shape of MLP architecture and its train parameters, listed in Table 1. The search space slightly differs from the grid search strategy by offering more resolution for the layer widths, learning rate, and over several seed values to compensate for the possibility of bad initialization. We sample 600 random points from this space to obtain an approximate distribution of the actual solution space for MLP architectures for each of the datasets. From this we can obtain the best model (MLP - RS_{best}) and the median model (MLP - RS_{median}).

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	Type	Grid search subspace	Random search subspace
Seed	int	36	[0,100]
Number of layers	int	{2, 3, 4}	[2, 8]
Layer width	int	{10, 15, 20, 25, 100}	[100, 1000]
Learning rate	float	5e-3	[1e-4, 1e-2]
Optimizer	operator	{SGD, RMSProp, Adam}	{SGD, RMSProp, Adam}
Activation function	function	{ReLU, ELU, SELU, tanh}	{ReLU, LReLU, ELU, SELU, tanh, Sin}
Train epochs	int	{20, 25, 50, 75}	50

4.3 Evolving Activation Functions

In this section, we describe in detail the setup for evolving activation functions.

4.3.1 Search space and Solution Encoding

The space of all feasible solutions is called a search space. Each activation function in the search space represents one possible solution. Every activation function is represented as a tree consisting of unary and binary operators with leaves corresponding to function inputs $\vec{x}$, namely 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 protected division, where the denominator value is replaced with $\epsilon = 10^{-4}$ if the denominator's absolute value is smaller 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 maximal tree depth is limited to 12. Moreover, there is no constraint of tree balancedness as discussed in [RZL17] and [BMM20].

4.3.2 Evolutionary Process

The selection process used in this work is presented as Algorithm 1 and employs a variant called steady-state tournament selection with the size of the tournament (k) equal to 3.

Algorithm 1 Steady-state tournament selection.

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;

Starting with a population of P randomly created activation functions, a tournament with three randomly selected individuals is performed in each iteration of the selection process. The worst individual from the tournament is eliminated, and a new one is constructed by performing crossover on the remaining two. The mutation is applied to the new individual, subject to the defined individual mutation rate. With P new individuals created in this way, a single generation of the evolutionary process is completed. The best activation function is always preserved in the population (elitism) since the best individual can never be selected for elimination. This process is repeated for a number of generations, and the activation functions with the best performance are returned as a result.

Evolution offers a more efficient approach to space sampling since it uses information from previous samplings to guide the search. We use this technique to find an activation function that optimizes the generalization of our models. The search space of functions is very large and requires some assumptions to make the search feasible. First, we assume the function can be represented as a tree, making the genetic programming technique a natural choice. Here, we constrain leaves to only be the function input, without constants or learnable parameters. Next, we limit the maximal depth of candidate trees to restrict the search to a fast to evaluate subspace of functions since they need to be evaluated for each layer. Finally, we limit the representation's expressivity by defining a set of possible unary and binary operations that can be used as function nodes (see Section 4.3.1). We use a population of 20 individuals and run the algorithm with a budget of 2000 evaluations.

4.3.3 Fitness Function

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

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

The goal is to minimize fitness value F, where the optimal value is 1 (this would require only a single trace to break the target, which represents the optimal scenario). The guessing entropy is averaged over 100 attacks on randomly selected data subsets. The maximum subset sizes were selected depending on a particular experiment to balance between differentiation of result qualities and computation time. In turn, this induces very similar results between individuals in initial iterations and slows the EA convergence. To remedy this, we add the accuracy error $(1 - accuracy)$ to the fitness, which is also subject to minimization, and adds additional information for differentiation between results. In preliminary experiments, we observed faster convergence by using this fitness function.

We note that it could be somewhat counterintuitive to use accuracy for SCA. This problem can be especially pronounced for the HW leakage model as it results in highly imbalanced data [PHJ+18]. Still, as the second part of the fitness function is bounded in the range $[0, 1]$, for neural networks that perform well (i.e., those that reach GE of 0 in a small number of traces), added information about accuracy can help by providing more search space gradient. When the number of required traces is high, the accuracy term's contribution is small, so the number of traces remains the primary objective.

4.3.4 Mutation and Crossover

In mutation, one node in an activation function tree is selected uniformly at random, and at that point, the selected subtree is replaced by generating a new subtree, respecting the maximal depth limit. The probability of mutation was selected from the preliminary experiments on the ASCAD fixed key dataset to provide reasonable exploration and exploitation properties and was kept at 70% through all of the experiments. In the crossover, two parent activation functions exchange randomly selected subtrees, producing new children activation functions. The crossover is performed with a simple tree crossover with 90% bias for functional nodes being selected as crossover points.

4.4 Learning System

All experiments were performed on a machine using a single GeForce GTX 1080 Ti graphics card, i7-6700 CPU, and 32 GB of RAM, running Ubuntu 16.04. We implemented our experiments in Python 3.7 with the usage of the DEAP[FDG+12] framework (v1.3.1) for evolutionary algorithms and PyTorch framework [PGM+19] (v1.7.0) for deep learning with the CUDA (v11.2) backend.

As our model architectures do not require a significant GPU memory, we fully utilize the hardware by parallelizing individuals' evaluation step via multiprocessing. This proved crucial as our experiments are extremely time-consuming, in the order of 7 days per architecture search and 14 days per evolution without parallelization. Note that the upper limit of the number of processes is dictated by the available GPU memory and physical CPU cores.

To ensure our experiments' reproducibility, we carefully set the seed value on both the CPU and GPU side. Python's *random*, *numpy*, and *torch* libraries provide methods for managing the state of a random generator. Implementing a simple context manager, entire blocks of an experiment can be run under the standardized setup while maintaining the code clean and automatically restoring the context afterward.

5 Results

This section presents the experimental results, demonstrating that evolved activation functions can outperform commonly used activation functions. We first search for the optimal network architecture and hyperparameters using the previously discussed architecture search methods (random search - RS and grid search - GS) for each experimental setup. Then on the selected setup, we apply the evolutionary algorithm to further optimize the model by changing its activation function. Finally, we compare the results of mentioned techniques on both datasets and leakage models. During the evaluation of GS and RS on ASCAD fixed key dataset with the ID leakage model, we truncated the evaluation of $\overline{Q}_{t_{GE}}$ to 1000 as we observed over 25% of results lied in this subspace, thus leading to an efficient region of interest. Additionally, we focused on 500 traces during the evaluation of GP to further improve efficiency while still obtaining better results. For the random keys dataset, we needed to increase the truncation bar to 1000 as it became more difficult to

Table 2: Final $\overline{Q}_{t_{GE}}$ values for ASCAD fixed and random keys datasets on the Hamming weight leakage model. Here 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}.

	Fixed Key			**Random Keys**		
	best	2nd best	3rd best	best	2nd best	3rd best
CNN - GS_{best}	299	-	-	606	-	-
CNN - GP	287	331	339	$\overline{Q}_{t_{GE}} > 5000$	$\overline{Q}_{t_{GE}} > 5000$	$\overline{Q}_{t_{GE}} > 5000$
MLP - RS_{best}	561	669	749	1133	1592	1615

Table 3: Final $\overline{Q}_{t_{GE}}$ values for ASCAD fixed and random keys datasets on the ID leakage model. Here we compare the best obtained value of grid search on the CNN model (CNN - GS_{best}), version with its evolved activation function (CNN - GP) and the best obtained MLP model on random search MLP - RS_{best}. The star denotes result obtained from reconstructing the resulting architecture in [ZBHV19].

	Fixed Key			**Random Keys**		
	best	2nd best	3rd best	best	2nd best	3rd best
CNN - GS_{best}	191*	-	-	$\overline{Q}_{t_{GE}} > 1000$	-	-
CNN - GP	115	123	130	$\overline{Q}_{t_{GE}} > 1000$	$\overline{Q}_{t_{GE}} > 1000$	$\overline{Q}_{t_{GE}} > 1000$
MLP - RS_{best}	156	162	191	145	163	194

obtain 25% results in this subspace. Finally, for all of the HW leakage model tasks, we further increased the bar to 5000 as they proved to be quite a bit harder.

In Tables 2 and 3, we depict the best-obtained results for the Hamming weight and the ID leakage models, respectively. We additionally compare with the state-of-the-art results [ZBD+20] when possible. The notation $\overline{Q}_{t_{GE}} > x$ denotes that we could not reach GE of 0 in x attack traces. We also depict the three best results obtained with various search techniques.

First, for the HW leakage model (Table 2) and fixed key, the best results are obtained with GP, while grid search with CNN performs slightly worse. Random search for MLP results are much worse than the first two, but we still manage to break the target. For random keys, we can see that the best results are reached for CNN with grid search, followed by MLP obtained through random search. Interestingly, CNN evolved with GP cannot converge even with 5000 attack traces. The architectures end with guessing entropy values: 108, 110, and 111 for the top 3 individuals, respectively. We postulate this happens as the random keys dataset is more difficult and becomes easier to overfit. Still, we believe adding more generations to the GP procedure would improve its behavior and attack results.

In Table 3, we depict results for the ID leakage model. Interestingly, for the fixed key, we see that both CNN evolved with GP and MLP with random search work better than related work [ZBHV19]. For random keys, we reach good results with MLP with random search only. Again, this shows that random keys setup is more difficult, and we require 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 respectively with values: 125, 128, and 128.

Next, in Table 4, we give median results for the HW leakage model when comparing MLP architectures obtained with random search and after evolving activations functions with GP. Note that GP improves the performance significantly for the fixed key, while GP

Table 4: Results of the EA effectiveness experiment for ASCAD fixed and random keys on 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
$\overline{Q}_{t_{GE}}$	2377	1168	3350	$\overline{Q}_{t_{GE}} > 5000$

Table 5: Results of the EA effectiveness experiment for ASCAD fixed and random keys datasets on 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

does not converge for the random keys dataset. At the same time, the configuration found with random search manages to break the target in somewhat more than 5000 attack traces, ending with a guessing entropy value of 124. As the results with GP denote that the median does not manage to break the target, this again reiterates that we require more than 100 generations to evolve good activation functions.

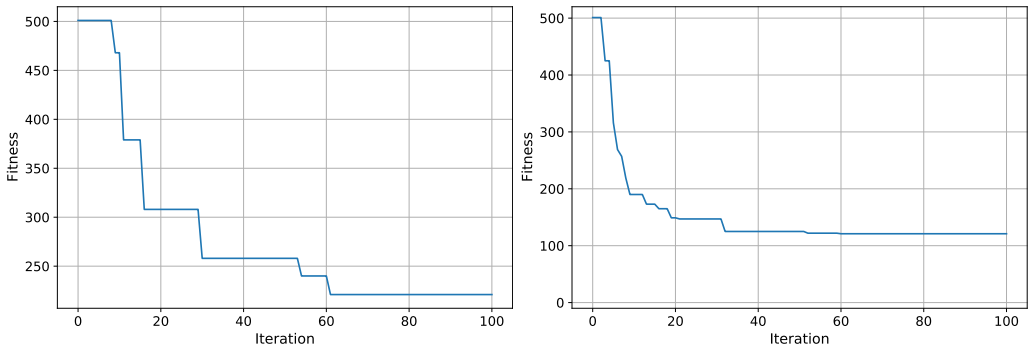
Finally, in Table 5, we compare the median results for MLP with random search and after using genetic programming when considering the ID leakage model. Observe how for both fixed and random keys settings, GP reaches significantly better results. This indicates that while the random search can find a good performing neural network architecture just by guessing, the average results obtained over a number of solutions are not very good. On the other hand, evolving customized activation functions manages to improve the neural network performance significantly.

5.1 ASCAD Fixed Key

For the HW leakage model, we obtained a CNN architecture with the tail of four hidden dense layers of width 100, activated by ELU, and initialized with seed 36. The training setup uses the $RMSProp$ optimizer with a learning rate of 0.005 over 20 epochs with batch size 64. For the architecture search of MLP, we obtained a model with eight hidden dense layers of width 478, activated by $ReLU$, and initialized with seed 42. The training setup uses the Adam optimizer with a learning rate of 0.0017 over 50 epochs with batch size 200.

For the ID leakage model, we reimplemented the CNN reported in [ZBHV19] and obtained similar results. The training setup uses the Adam optimizer with a learning rate of 0.005 over 50 epochs with batch size 50. The architecture is initialized with a seed equal to 36 and uses the $SELU$ activation function. For the architecture search of MLP, we obtained a model with five hidden dense layers of width 661, activated by the $Sine$ function and initialized with seed 2. The training setup uses the Adam optimizer with a learning rate of 0.0086 over 50 epochs with batch size 200.

The best activation functions we obtained are denoted below, with the first letter in the subscript corresponding to the setup type and the second one to the architecture type. Note that the functions look rather complex, and it would be hard to expect that a human designer would find them. Still, the experimental results show they work very well.



(a) Fitness value of MLP - GP. **(b)** Fitness value of CNN - GP.

Figure 1: The evolution of fitness value for the ASCAD fixed key dataset and the ID leakage model.

$$a_{ID,C}(\vec{x}) = sin(erf(\vec{x} - softmin(\vec{x})^2)) - \tag{4}$$

$$\vec{x}^2 \cdot softsign(\sigma(softmin(softsign(\sigma_H(\sigma(normalized(softmax(\vec{x}^2))))))))) \tag{5}$$

$$a_{ID,M}(\vec{x}) = normalized(softsign(normalized(\vec{x} \cdot tanh(softplus(softsign(erf(\vec{x})))))))) \tag{6}$$

$$a_{HW,C}(\vec{x}) = -(tanh(-\vec{x}) \cdot |softsign(2\vec{x}) - (\sigma(tanh(\vec{x})) + ELU(\vec{x}))|) \tag{7}$$

$$a_{HW,M}(\vec{x}) = softmax(softmin(sin(normalized(tanh(\frac{-1}{\sigma(\vec{x}^2)^2})))) - \vec{x}) \tag{8}$$

$$\tag{9}$$

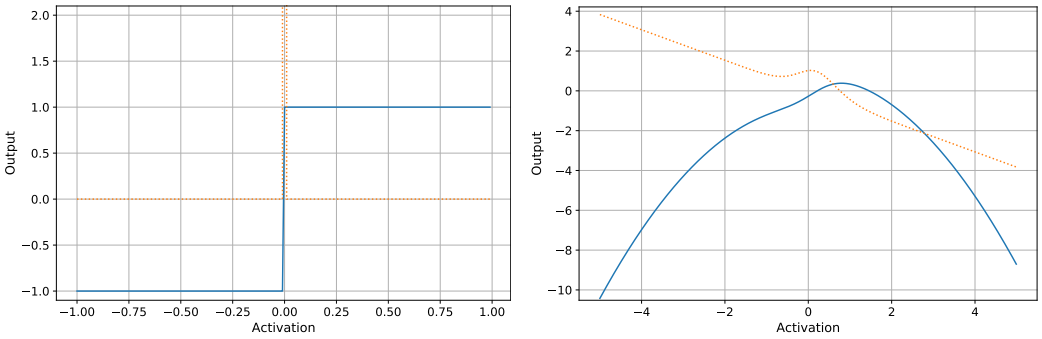
In Figure 1, we depict the GP evolution convergence plots for the MLP and CNN architecture for the ASCAD fixed key dataset. Notice how both architectures improve with iterations (clearly showing there is learning happening). Especially strong convergence can be seen for CNN, where the final fitness is more than twice smaller than in the MLP case.

Next, in Figure 2, we depict the best-obtained activation function and its derivation. Recall, the derivation is important as we require the differentiable function if we use the backpropagation algorithm, as is common in the training of neural networks. Notice that we obtain a non-monotonic function for CNN, which is strikingly different from commonly (and state-of-the-art) used activation functions.

Finally, in Figure 3, we depict the number of attack traces required to reach a guessing entropy of 0. That value is denoted with a red dot. Notice that we break the target significantly faster when using CNN than MLP, but both techniques perform well. This ensures that we can find custom activation functions for SCA that perform well regardless of the neural network selection. Moreover, in Appendix A, we depict the corresponding plots for the HW leakage model.

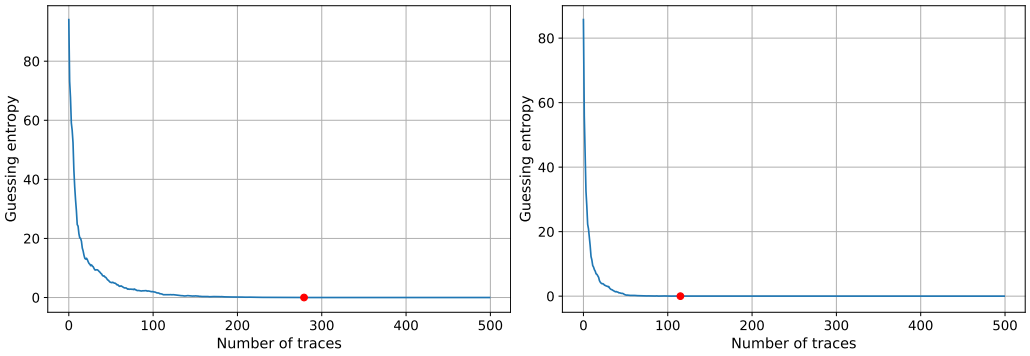
5.2 ASCAD Random Keys

In the case of the HW leakage model for the random keys dataset and CNN's architecture search, we obtained an architecture tail with four hidden dense layers of width 100, activated by the SELU activation function, and initialized with seed 36. The training setup uses the SGD optimizer with a learning rate equal to 0.005 over 75 epochs with batch size 50. Using the architecture search of MLP, we obtained a model with two hidden dense layers of width 716, activated by SELU and initialized with seed 2. The training setup uses an SGD optimizer with a learning rate of 0.0042 over 50 epochs with batch size 200.



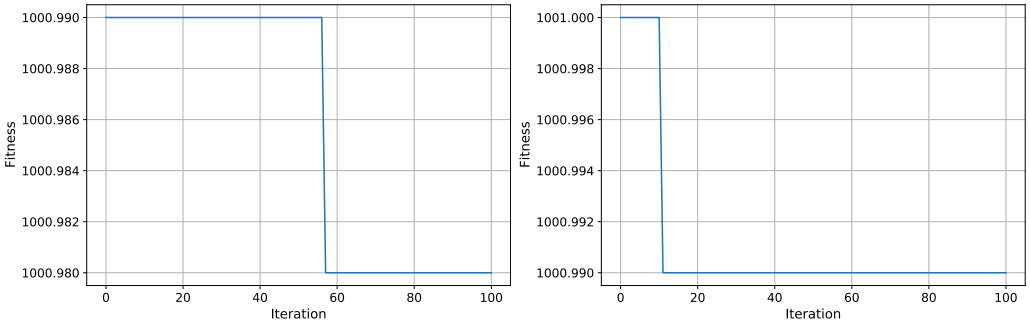
(a) Best activation function for MLP - GP. **(b)** Best activation function for CNN - GP.

Figure 2: Plots represent the 1D slice of the best activation function (solid) and its derivative (dotted) obtained through evolution on the ASCAD fixed key dataset for ID prediction. 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 (Table 5). **(b)** The guessing entropy of CNN - GP (Table 3).

Figure 3: The guessing entropy of the best evolved activation functions on the ASCAD fixed key dataset and the ID leakage model.



(a) Fitness value of MLP - GP. **(b)** Fitness value of CNN - GP.

Figure 4: The evolution of fitness value on the ASCAD random keys dataset and the ID leakage model.

For the ID leakage model and the CNN's architecture search, we obtained an architecture tail with two hidden dense layers of width 100, activated by *tanh* activation function and initialized with seed 36. Using the architecture search of MLP, we obtained a model with five hidden dense layers of width 622, activated by *ReLU* and initialized with seed 2. The training setup uses the Adam optimizer with a learning rate of 0.00086 over 50 epochs with batch size 200.

The activation functions we obtained are shown below, with the first letter in the subscript corresponding to the setup type and the second one to the architecture type. Notice that these activation functions are slightly less complex (having fewer terms) than in the ASCAD fixed key scenario.

$$a_{ID,C}(\vec{x}) = (tanh(|sin(cos(\vec{x}))|))^{-1} \tag{10}$$

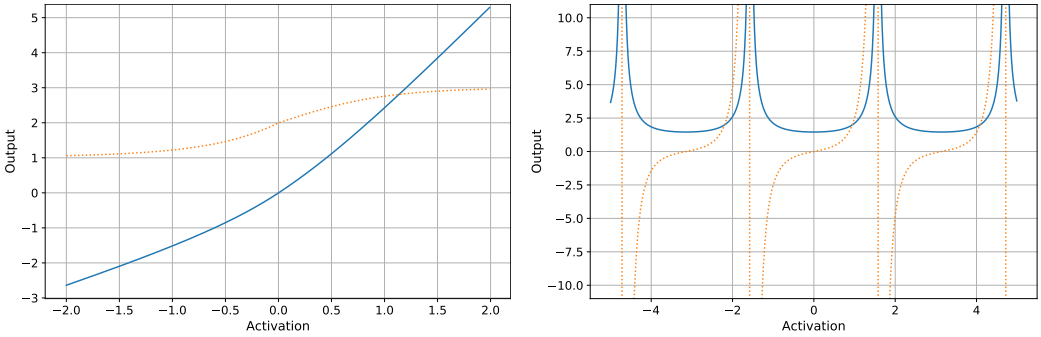
$$a_{ID,M}(\vec{x}) = softplus(\vec{x} + ELU(\vec{x})) + \vec{x} - \sigma(ELU(cos(\sigma_H(cos(softplus(\sigma(\vec{x})))))))) \tag{11}$$

$$a_{HW,C}(\vec{x}) = exp(cos(softmax(sin(\vec{x})^{-1}))) \tag{12}$$

$$a_{HW,M}(\vec{x}) = ReLU(\sigma_H(cos(softmin(normalized(|softmin(softmax(\vec{x}))|))^2))) \tag{13}$$

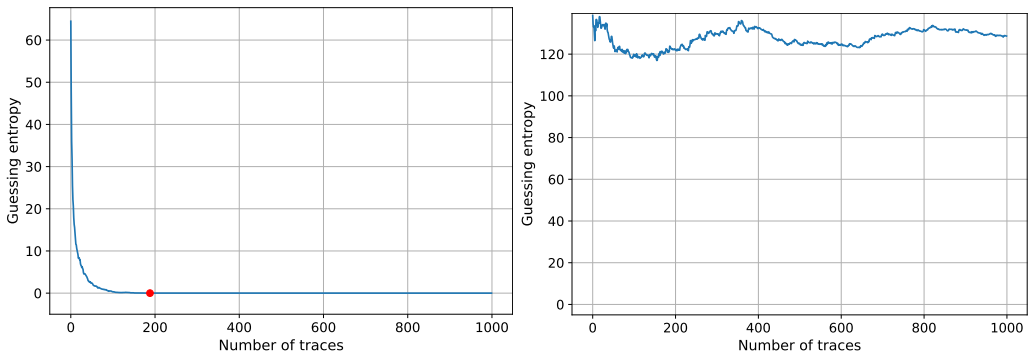
Next, in Figure 4, we depict the convergence plots. Again, CNN converges faster, and even with a relatively short number of iterations, there is no more improvement in the fitness value. Considering rather small improvements in the fitness value, we could conclude that the evolution process gets stuck in local optima. One potential solution could be to consider larger mutation rates to stimulate search space exploration.

Figure 5 presents plots for the activation function and its derivation. Notice that while for MLP, the obtained functions have some similarity with commonly used ones, for CNN, the shape is rather unusual (and not intuitive that it would work). Finally, in Figure 6, we depict the guessing entropy results. Interestingly, MLP works very well (on the level as for the ASCAD with fixed key dataset), while CNN does not converge. Again, this reiterates that it is easier to tweak MLP architectures, regardless of whether it is done via hyperparameter tuning [WPP20] or evolution of activation functions as investigated here. What is more, the results indicate that MLP architectures are sufficient to break the targets, especially if there is no trace misalignment (recall that in this work, we consider only synchronized traces). For CNNs, working with larger mutation rates and longer evolution could resolve the problems indicated here.



(a) Best activation function for MLP - GP. **(b)** Best activation function for CNN - GP.

Figure 5: Plots represent the 1D slice of the best activation function (solid) and its derivative (dotted) obtained through evolution on the ASCAD random keys dataset. 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 (Table 5). **(b)** The guessing entropy of CNN - GP (Table 3).

Figure 6: The guessing entropy of best evolved activation functions on the ASCAD random keys dataset.

6 Conclusions and Future Work

This paper investigates how neuroevolution can improve deep learning-based side-channel analysis. More precisely, we consider the setting where genetic programming evolves activation functions specifically adapted for the 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 the attack behavior.

We observe that activation function evolution has higher efficiency for a simpler dataset, indicating that more work is needed to understand this approach's advantages and drawbacks. Additionally, we observe the need for more informative and cost-effective fitness functions that would lead to better individuals faster.

Since this is the first work considering neuroevolution for SCA, there are many possible research directions for future work. One direction would be to consider evolving other elements of the learning procedure, like the loss function. Another option could be to use neuroevolution to evolve the whole neural network architecture, not restricting to activation function only. Finally, another viable option is to use evolution instead of the backpropagation algorithm so that the activation function search does not need to be restricted to differentiable elements. Considering the application of different activation functions, one natural option would be to consider the evolution of activation functions that are different for various layers. It would also be interesting to investigate how transferable the obtained activation functions are when considering already designed neural network architectures. Our preliminary testing indicates this transferability to be rather limited, but more experiments are required.

References

[BMM20] Garrett Bingham, William Macke, and Risto Miikkulainen. Evolutionary optimization of deep learning activation functions. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 289–296, New York, NY, USA, 2020. Association for Computing Machinery.

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

[BR18] Mina Basirat and Peter M. Roth. The quest for the golden activation function. *CoRR*, abs/1808.00783, 2018.

[CB04] Ronan Collobert and Samy Bengio. Links between perceptrons, mlps and svms. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 23, New York, NY, USA, 2004. Association for Computing Machinery.

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

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

[CUH16] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[CXWT19] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[DJ99] Wlodzislaw Duch and Norbert Jankowski. Survey of neural transfer functions. *Neural Computing Surveys*, 2:163–213, 1999.

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey, 2019.

[FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[FDG+12] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[FL18] Chelsea Finn and Sergey Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm, 2018.

[FSH15] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 1128–1135. AAAI Press, 2015.

[Fuk80] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GM20] Santiago Gonzalez and Risto Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization, 2020.

[HMA17] Alexander Hagg, Maximilian Mensing, and Alexander Asteroth. Evolving parsimonious networks by mixing activation functions, 2017.

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

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society.

[KH19] D. Klabjan and M. Harmon. Activation ensembles for deep neural networks. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 206–214, 2019.

[KJJ99a] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KJJ99b] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, UK, 1999. Springer-Verlag.

[KJJR11] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1(1):5–27, 2011.

[KMK15] Kishore Konda, Roland Memisevic, and David Krueger. Zero-bias autoencoders and the benefits of co-adapting features, 2015.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

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

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 972–981, Red Hook, NY, USA, 2017. Curran Associates Inc.

[LB95] Yann Lecun and Yoshua Bengio. *Convolutional networks for images, speech, and time-series*. MIT Press, 1995.

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

[LY96] Y. Liu and X. Yao. Evolutionary design of artificial neural networks with different nodes. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 670–675, 1996.

[MHN13] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[MHR+18] Alberto Marchisio, Muhammad Abdullah Hanif, Semeen Rehman, Maurizio Martina, and Muhammad Shafique. A methodology for automatic selection of activation functions to design hybrid deep neural networks, 2018.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[MOP06] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, December 2006. ISBN 0-387-30857-1, http://www.dpabook.org/.

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

[MR18] F. Manessi and A. Rozza. Learning combinations of activation functions. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 61–66, 2018.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

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

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PH18] Christophe Pfeifer and Patrick Haddad. Spread: a new layer for profiled deep-learning side-channel attacks. Cryptology ePrint Archive, Report 2018/880, 2018. https://eprint.iacr.org/2018/880.

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

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

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

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[RWPP21] Jorai Rijsdijk, Lichao Wu, Guilherme Perin, and Stjepan Picek. Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2021/071, 2021. https://eprint.iacr.org/2021/071.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

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

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

[WPP20] Lichao Wu, Guilherme Perin, and Stjepan Picek. I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/1293, 2020. https://eprint.iacr.org/2020/1293.

[WRP19] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search, 2019.

[ZBD+20] Gabriel Zaid, Lilian Bossuet, François Dassance, Amaury Habrard, and Alexandre Venelli. Ranking loss: Maximizing the success rate in deep learning side-channel analysis. 2021:25–55, Dec. 2020.

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

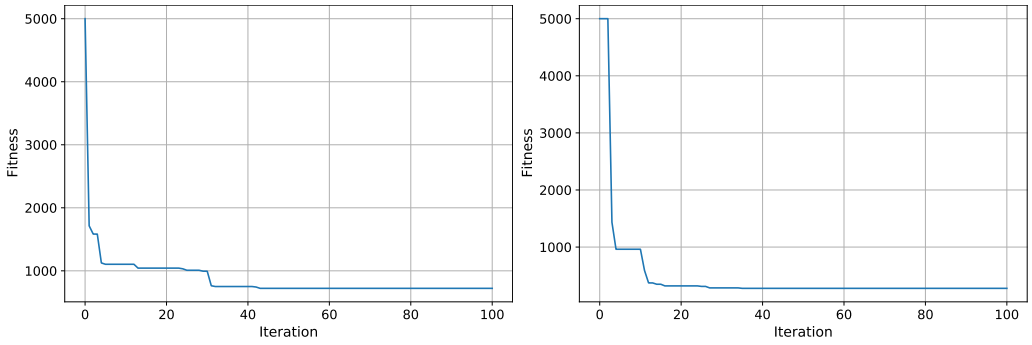
[ZL17] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2017.

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

A ASCAD Fixed Key for the HW Leakage Model

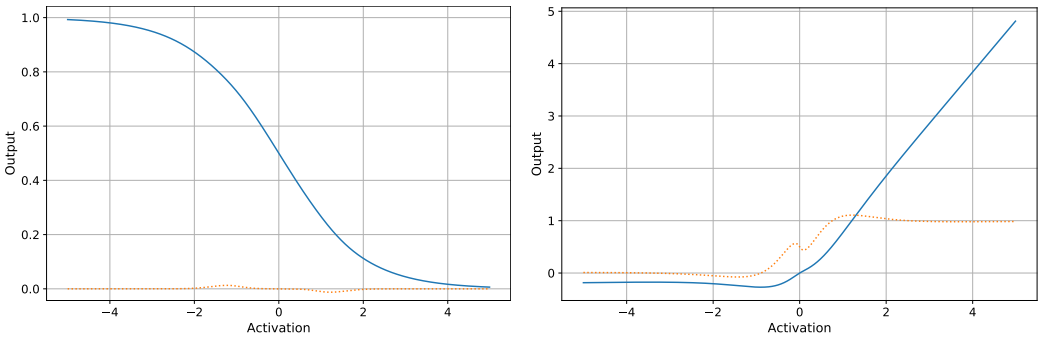
For the HW leakage model and the ASCAD fixed key dataset, we depict in Figure 7 the convergence plots for the MLP and CNN architectures. We see that CNN converges faster, where the final fitness is two times smaller than in the MLP case. Next, in Figure 8, we depict the best-obtained activation function and its derivation. Both obtained functions have similarities with the commonly used ones. Interestingly, the GP evolved activation functions with similar properties to $tanh(-x)$ (see Figure 8a) function and ELU (see Figure 8b) activation function.

Finally, in Figure 9, we depict the number of attack traces required to reach a guessing entropy of 0, and we denoted that value with a red dot. Despite the fact that MLP and CNN architectures perform well, we break the target significantly faster when using CNN than MLP.



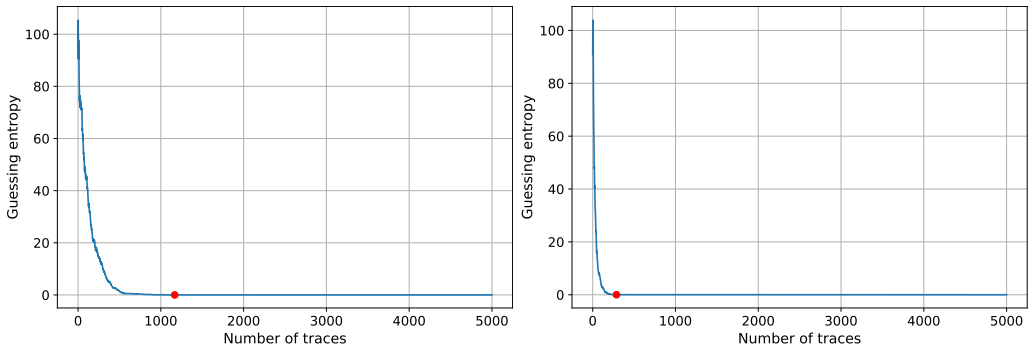
(a) Fitness value of MLP - GP. **(b)** Fitness value of CNN - GP.

Figure 7: The evolution of fitness value on the ASCAD random keys dataset and the HW leakage model.



(a) Best activation function for MLP - GP. **(b)** Best activation function for CNN - GP.

Figure 8: Plots represent the 1D slice of the best activation function (solid) and its derivative (dotted) obtained through evolution on the ASCAD fixed key dataset. 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 (Table 5). **(b)** The guessing entropy of CNN - GP (Table 3).

Figure 9: The guessing entropy of best evolved activation functions on the ASCAD fixed key dataset.