Limitations of the Use of Neural Networks in Black Box Cryptanalysis

Limitations of the Use of Neural Networks in Black Box Cryptanalysis

Emanuele Bellini[1][0000−0002−2349−0247], Anna Hambitzer[1], Matteo Protopapa[2], and Matteo Rossi[2]

[1] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
{name.lastname}@tii.ae
[2] Politecnico di Torino, Torino, Italy
{name.lastname}@polito.it

Abstract. In this work, we first abstract a block cipher to a set of parallel Boolean functions. Then, we establish the conditions that allow a multilayer perceptron (MLP) neural network to correctly emulate a Boolean function. We extend these conditions to the case of any block cipher. The modeling of the block cipher is performed in a black box scenario with a set of random samples, resulting in a single secret key chosen plaintext/ciphertext attack. Based on our findings we explain the reasons behind the success and failure of relevant related cases in the literature. Finally, we conclude by estimating what are the resources to fully emulate 2 rounds of AES-128, a task that has never been achieved by means of neural networks. Despite the presence of original results and observations, we remark the systematization of knowledge nature of this work, whose main point is to explain the reason behind the inefficacy of the use of neural networks for black box cryptanalysis.

Keywords: black-box · cryptanalysis · neural networks · cipher emulation · AES

1 Introduction

The similarities between finding the cryptographic key of a symmetric cipher and finding the unknown weights of a neural network have been known since long time. See for example Rivest's survey at Asiacrypt 1991 [1] and references therein. Due to the impressive and constant progress of technology, the adoption of neural networks is becoming increasingly popular and effective in solving more and more complex problems. This success of neural networks has tempted many cryptographers to exploit them for cryptanalysis. While there are several ways of using neural networks and, more in general, machine learning in conjunction with cryptography, we want to focus our attention on the use of neural networks in the context of *black box cryptanalysis*. The black box approach attempts to cryptanalyze a family of symmetric ciphers by only interrogating an oracle which can compute plaintext/ciphertext pairs coming from a specific instantiation of this family. No other information are allowed to the attacker, hence the name "black box". If a family of ciphers can be attacked in the black box scenario this

implies that these ciphers are not suitable for practical applications. The most popular ciphers are believed to be secure under this scenario, and, moreover, are even secure in weaker scenarios, where the knowledge of the internal structure of the cipher is accessible by the attacker. Intuitively, being secure in a weaker scenario gives little hope of finding a complete break in a stronger scenario such as the black box one. In spite of this maybe simplistic intuition, we can count numerous attempts of using neural networks to either distinguish the output of a cipher from that of a random function, or to discern the output of different cipher families, or to emulate, or, the hardest case, to even recover the key of a particular cipher instance. However, to date none of these attempts has outperformed existing conventional cryptographic attacks. In this work, we provide insights on why using neural networks in black box cryptanalysis gives little hope of success. We would like to stress that in this work we do not consider cryptanalysis techniques based on the knowledge of the internal structure of the cipher.

The remainder of this paper is structured as follows. After a brief introduction on neural network terminology and basic notions regarding Boolean functions, block ciphers and how the latter can be abstracted by the former (section 2), we speculate on the hardness of emulating a random Boolean function and, consequently, a block cipher (section 3). We analyze prior works on the subject under the light of this abstraction (section 4). We support with experimental evidence our claims on the hardness of emulating Boolean functions (section 5). Finally, in the light of the developed theory, we estimate the resources needed to fully emulate 2 rounds of AES (section 6), a task that has never been performed by neural networks.

2 Preliminaries

In this section we introduce basics of neural networks for black box cryptanalysis, Boolean functions, and how block ciphers can be defined in terms of Boolean functions.

2.1 Neural Networks

We refer the interested reader to educative theoretical [2] and practical [3] introductions to neural networks and the field of deep learning. Here, we concentrate on condensed explanations of concepts elementary for understanding the following sections of this work.

The neural networks usually applied in cryptanalysis are MLP, LSTM and CNN networks (see Table 1). MLP, LSTM and CNN refer to *multilayer perceptron*, *long-short term memory* and *convolutional neural network*, respectively. All three network types contain *artificial neurons* organized in *layers* and "learn" by adjusting a set of *trainable parameters*: the *weights* w with which the neurons are connected to each other and a so-called *bias value* b for each neuron. During the learning phase the network is presented with a *training dataset*. The success of learning is quantified by the network's performance on previously unseen samples from the *validation dataset*, i.e. the goal of learning is *generalization* [2]. To achieve such generalization, deep learning is concerned with the identification of

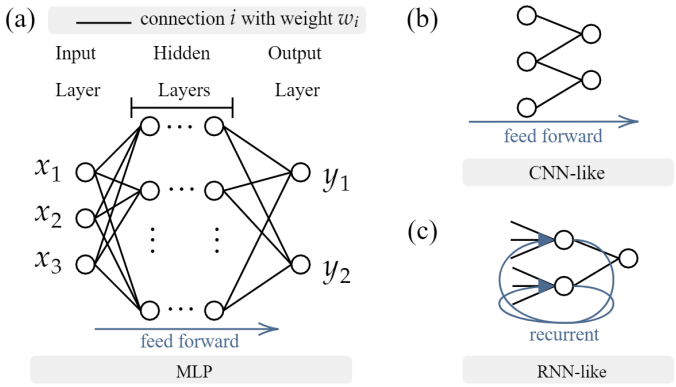

Fig. 1: (a) Example of a multilayer perceptron (MLP) architecture: The neurons at the input layer receive x_1, x_2, x_3. Each neuron is connected with weights w_i to all neurons in the following layer. The input signals cause a feed-forward activation to propagate through the hidden layers and produce the outcomes y_1, y_2 in the output layer. (b) CNN-like layers consist of neurons which are connected only to neurons within their receptive field in the previous layer. (c) In a recurrent neural network (RNN) layers receive an input which depends on their output at a previous time. The LSTM cell is a well known representative of a recurrent structure.

the main features which *represent* a concept [2]. The respective representational features are worked out by one or multiple *hidden layers*. Training a neural network is most commonly achieved by applying a *backpropagation algorithm* [4]. First, a *batch* of training samples is presented at the input of the neural network. The consequent activations propagate through the neural network, resulting in a signal in the output layer. In a supervised setting, this output signal can be compared to the known *labels* of the training samples. The distance of the output signal to the known label is quantified by a *loss function*. During backpropagation the contribution of each neuron's parameters to the total loss of the loss function is evaluated and the network parameters are adjusted by an *optimizer*, aiming at a minimal loss after each batch. Typical optimizers are the *gradient descent* and its advanced variants. The step size taken during the gradient descent is determined by the *learning rate*. Once all training samples have been presented to the neural network, one *epoch* is over. Often, training involves several hundred epochs and batch sizes can vary between 1 (*stochastic gradient descent*) to the full size of the training data set (*deterministic gradient descent*).

Figure 1 illustrates the differences between MLP, LSTM and CNN. The MLP constitutes the *"quintessential example of a deep learning model"* [2]. Here, each neuron in one layer is connected to all neurons in the next layer in a feed-forward manner. The MLP can have a single or multiple hidden layers. The CNN, like the MLP, is a feed-forward network. However, its design is motivated by the

mammalian visual cortex and each neuron is only connected to neurons in its receptive field in the previous layer. In contrast to the feed-forward structure of MLP and CNN, the LSTM contains so-called *recurrent* connections between the neurons. The output of the neuron becomes dependent on a past state of the network which leads to a kind of "memory" [5].

Essential to all network types is the introduction of nonlinearity in the form of an *activation function* $a(w, b)$ which determines the output of each neuron in a single layer. It can be shown that the introduction of nonlinearity in the activation function, as well as minimally one hidden layer leads to *universality* of the neural network in its ability to model any continuous [6,7,8] function. In general, the representational power [9] will rise with increasing depth (i.e. the number of layers) and width (i.e. number of neurons in a single layer) of the neural network. Practically, however, the problem of successfully training a neural network with sufficient representational power can still be NP-hard [10], resulting in unmanageable training time.

2.2 Boolean functions and block ciphers

Block ciphers For simplicity, in this work we will focus our attention only on block ciphers, but all our arguments can be easily extended to other symmetric ciphers such as hash functions, stream ciphers or cryptographic permutations.

Let $\mathcal{M}$ (*plaintext/ciphertext space*) and $\mathcal{K}$ (*key space*) be the set of b-bit and κ-bit vectors, respectively. Note that b is usually called the block size, while κ the key size. A block cipher $E_k(x) : \mathbb{F}_2^b \times \mathbb{F}_2^\kappa \mapsto \mathbb{F}_2^b$ is a family of permutations over the plaintext/ciphertext space $\mathcal{M}$. Each permutation of the family is indexed by a key $k \in \mathcal{K}$. Modern block ciphers are built by composing several times a round function taking as input the current state of the cipher and a round key that is derived from the master key k by means of an algorithm called key schedule. So the block cipher operation can be expressed as $E_k^r(x) = R_{k_r} \circ R_{k_{r-1}} \circ \cdots \circ R_{k_1}$, where $(k_1, \ldots, k_r) = \mathsf{KeySchedule}(k)$. While a specific block cipher is usually defined for a precise number of rounds r, to assess their security, it is common to study *reduced-round* ciphers. Alternatively, cryptographers also consider scaled versions of ciphers, i.e. ciphers performing similar operations but with respect to a smaller block and key size.

Block ciphers as Boolean functions Each ciphertext bit of a block cipher can be defined by a Boolean function whose variables represents the plaintext and key bits. More precisely, the i-th bit of the ciphertext can be expressed as:

$$f_i(x_1, \ldots, x_b, k_1, \ldots, k_\kappa) = \sum_{(v_1,\ldots,v_b)\in\mathbb{F}_2^b} c_{(v_1,\ldots,v_b)}(k_1, \ldots, k_\kappa) x_1^{v_1} \cdots x_b^{v_b}, \quad (1)$$

where $c_{(v_1,\ldots,v_b)}(k_1, \ldots, k_\kappa) = \sum_{(v'_1,\ldots,v'_\kappa)\in\mathbb{F}_2^\kappa} a_{(v'_1,\ldots,v'_\kappa)}^{(v_1,\ldots,v_b)} k_1^{v'_1} \cdots k_\kappa^{v'_\kappa}$. Note that once the key $k = (k_1, \ldots, k_\kappa)$ is fixed, each f_i is a Boolean function of degree at most b with at most 2^b coefficients. When uniformly sampling a Boolean function f from the set of all Boolean functions over b variables, f will have on average 2^{b-1} nonzero coefficients. A secure cipher should be such that the Boolean functions

representing its output bits appear uniformly sampled. For real ciphers, b is at least 64 bits (128, 192, or 256 are also very common), which makes it impossible to even list all the coefficients of the Boolean function representing one output bit. On the other hand, the Boolean function representing the output of a single round (with respect to the input bits of the round) does not look random in general. In particular, one output bit of the round function usually depends on only some of the input bits. As we will explain in the upcoming sections, we believe this property to be crucial in explaining the success and failure of previous works.

3 On the hardness of emulating Boolean functions

In this section we first recall some of the main works that are related to the hardness of learning Boolean functions. We then provide further motivations on why it is hard to model Boolean functions, especially in cryptographic scenarios.

3.1 Related work

The problem of learning Boolean circuits by means of neural networks has been extensively studied by the machine learning community. On the other hand, we are aware of only few direct applications of such results in cryptographic scenarios. For example, already in the early nineties, Kearns [11, Chapter 7] showed that the Boolean circuits representing some trapdoor functions used in asymmetric cryptography (such as RSA function) are hard to learn in a polynomial time. A similar hardness result was demonstrated in the work of Goldreich et al. [12] for the class of random functions. Indeed, in spite of these negative results, the attempts of modeling symmetric ciphers by means of neural networks are numerous, as we show in section 4.

Many works analyze what is the largest family of Boolean functions that can be modelled by a single neuron. For example, Steinbach and Kohutin [13] show that, using a polynomial as transfer function, a single neuron is able to represent a non-monotonous Boolean function. They also show how to decrease the number of inputs in the neural network by encoding the binary values of the Boolean variables as integers. Finally, they also propose an algorithm to compute the minimal number of neurons. In [9], Antony studies which type of Boolean functions a given type of single or multi neuron network (using either threshold, sigmoid, polynomial threshold, and spiking neurons) can compute, and how extensive or expressive the set of such computable functions is. Among these results, he shows that any Boolean function with m variables can be modelled by a neural network with a single hidden layer of 2^m neurons with threshold activation function [9, Theorem 3.9]. Indeed, only $\Omega(2^m/m^2)$ neurons are sufficient.

In general, even if any function that can be run efficiently on a computer can be modelled by a deep neural network, the learning procedure can be computationally hard [14]. It is an important open problem to understand if there exists properties of the data distributions that can facilitate the training phase. As an example of works in this direction, Malach and Shalev-Shwartz [15] show that the correlation between input bits and the target label affects the learnability of a Boolean function. Following this line, in appendix D we analyze the

dependence of the learning rate and certain cryptographic properties of Boolean functions.

3.2 Block ciphers and permutations

Let us consider the simplest block cipher, taking 1 bit input, 1 bit key and 1 bit output: $y_0 = E_{k_0}(x_0)$. Once the key is fixed, the block cipher is a permutation over the set of messages, in this case, the set $0, 1$. The only possible permutations are the identity and the bitflip. The permutations can be indexed by the value of the key k_0. Let us now consider the 2-bit block cipher, with a 2 bit input, 2 bit key and 2 bit output: $(y_0, y_1) = E_{(k_0,k_1)}(x_0, x_1)$. Once the key is fixed, the block cipher is a permutation over the set of messages, in this case, the set $\{00, 01, 10, 11\}$. The number of possible permutations over a set of 4 elements is $4! = 24$.

The permutations are represented by the concatenation of two Boolean functions.

Notice that with 2 bits we only have 4 possible values of the key, which means we cannot represent all possible permutations over the set $\{00, 01, 10, 11\}$ with a 2 bit key.

When we consider a 3-bit cipher the permutations are $8! = 40320$, and only $2^3 = 8$ of them can be indexed by a 3 bit key. For the three bit cipher, we finally have permutations that are represented by nonlinear Boolean functions. In principle, it is possible to compute the Boolean functions representing the output bits of a full real cipher. The problem is that, with this method, one has to know the outputs of all possible inputs, which, for example for AES-128, are 2^{128}. For a reduced-round cipher (i.e. a cipher that does not use all the rounds it was designed to use), it is possible that a single output bit is not influenced by all input bits, but only by a subset of them of size m. In this case, the Boolean function will have $O(2^m)$ coefficients.

3.3 Emulating the behaviour of a Boolean function

Without knowing the entire truth table or, equivalently, the entire set of coefficients, it is impossible to reconstruct the remaining missing values of a randomly selected Boolean function.

In appendix C, by means of a tiny example, we give an intuition of how one can measure the accuracy of an algorithm guessing the missing values of a randomly selected Boolean function.

Types of accuracy In general, we can define the following types of accuracy:

1. *m'-ary (or block) accuracy*: measuring how many output blocks are fully guessed correctly in the validation phase. In other words, we consider a sample guessed correctly if and only if all its bits match with the correct output. To compute the accuracy, divide the counter by the total number of m-bit output that have been guessed;
2. *Relative binary or (bit per block) accuracy*: measuring how many bits per output block are guessed correctly in the validation phase. To compute the accuracy, divide the counter by the total number of m-bit output that have been guessed;

3. *Absolute binary (or bit) accuracy*: measuring how many output bits are fully guessed correctly in the validation phase. To compute the accuracy, divide the counter by the product of the number of bits per block (2) and the total number of m-bit output that have been guessed.

Randomly guessing the output of a set of Boolean functions We now provide the probabilities of randomly guessing the output of a Boolean function in three different scenarios, which corresponds to three different ways of measuring the accuracy of a neural network.

Proposition 1. *Consider a set of m' Boolean functions with m variables. Suppose the value of t m-bits inputs is known for each function. The probability of randomly guessing correctly all m' bits for each of t' new outputs is given by $1/2^{t'm'}$.*

Proposition 2. *Consider a set of m' Boolean functions with m variables. Suppose the value of t m'-bits outputs is known for each function. The probability of randomly guessing correctly at least s bits of a new m'-bit output is given by $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}}$. The probability of randomly guessing correctly at least s bits for each m'-bit output for t' new output is given by $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}} t'$.*

Proposition 3. *Consider a set of m' Boolean functions with m variables. Suppose the value of t m-bits inputs is known for each function. The probability of randomly guessing correctly at least s bits among t' new m'-bit outputs is given by $\frac{\sum_{k=s}^{t'm'} \binom{t'm'}{k}}{2^{t'm'}}$.*

Note that in all previous propositions, the value of t and m do not appear in the probability. This is because, for a random Boolean function, the output bits of its truth table are uniformly distributed, and knowing part of the truth table, does not give any information about the missing part. On the other hand, if the guessing algorithm had some extra information about the Boolean functions, for example it knew that the output has to form a permutation, this probabilities could be improved. Unfortunately, we are not aware of how to incorporate the structure of a permutation over $\mathbb{F}_2^m$ into a neural network. Similarly, these probabilities might be lower if the Boolean function representing one output bit only depends on $\tilde{m}$ of the m input variables (as for a cipher that is not ideal, e.g. a reduced-round cipher). In this case, $2^{\tilde{m}}$ samples might be enough to train the network so that it can fully emulate the Boolean function. We analyze this case in Experiment 1 of section 5.

Trained neural networks are no better than random guessing One is interested in checking if a trained neural network can correctly predict new inputs better than an algorithm guessing uniformly at random would do. In our case, the block accuracy of the network should be higher than $1/2^{t'm'}$, the relative binary accuracy should be higher than $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}} t'$, and the absolute binary accuracy should be higher than $\frac{\sum_{k=s}^{t'm'} \binom{t'm'}{k}}{2^{t'm'}}$.

Conjecture 1. Let $\mathcal{N}$ be a multi-layer perceptron with m binary inputs and m' binary outputs. Suppose $\mathcal{N}$ has been trained with $t < 2^m$ samples, taken from m' parallel Boolean functions. Then we claim that the validation accuracy of the neural network cannot be better than the accuracy of an algorithm that uniformly guesses new outputs. More precisely,

1. the validation block accuracy measured over t' new samples is $1/2^{t'm'}$
2. the validation relative binary accuracy measured over t' new samples is $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}} t'$
3. the validation absolute binary accuracy measured over t' new samples is $\frac{\sum_{k=s}^{t'm'} \binom{t'm'}{k}}{2^{t'm'}}$

We give experimental evidence of the above conjecture in section 5. Also, in the remaining part of the manuscript, we will only consider the absolute binary accuracy, and we will refer to it as simply the *binary accuracy*.

3.4 Noisy bits

Because of what we explained in the previous section, training a neural network to fully model a block cipher is exponentially hard. In particular, for an n-bit block cipher in which each output bit depends on all n input bits, the cost of the training is $O(2^n)$ ($n = 128$ for the case of AES-128). On the other hand, for a reduced number of rounds, it is possible (especially in the early rounds), that each output bit only depends on $m < n$ input bits. If an adversary knew the position of the m input bits, it could train a network with only m inputs in time $O(2^m)$. We call *noisy bits* those $n - m$ bits for which the output does not depend on. For example, after 2 rounds of AES-128, each output bit depends on $m = 32$ input bits, and has 96 noisy bits. Unfortunately, in a black box scenario, the attacker has no knowledge about the position of the noisy input bits, so it is forced to use a neural network with n inputs. Suppose we are interested in modeling a single output bit. In this case, the neural network needs to understand which are the $n - m$ noisy bits that are not influencing the output bit. As we show in Experiment 2, it turns out that the complexity of the training increases exponentially with the number of noisy bits.

4 Analysis of previous results

In this section we analyze previous attempts of modelling symmetric ciphers in the black box scenario by means of neural networks.

By *black-box neural cryptanalysis* (or direct attacks with no prior information), we mean attacks that can be performed on any cipher, regardless of the cipher structure, except the input/output/key size. This type of attacks can be divided in attacks that aim at 1. distinguishing the output of the cipher from the output of another cipher, or distinguishing the output of the cipher from a random bit string; 2. emulating the behaviour of a cipher; 3. finding the key of the cipher. Of course, an attacker able to perform item 3 can also perform item 2, and being able to perform item 2 implies being able to perform item 1.

Usually, these kind of attacks are performed in the chosen plaintext scenario, so the attacker is given access to an oracle that can provide plaintext-ciphertext pairs encrypted under a certain key only known by the oracle. Furthermore, the attack is repeated for several keys, and in the case of key recovery, a new key (different from the ones used in the training) needs to be predicted. Here we describe in details only the previous attempts of cipher emulation, since they are relevant to our work. In Table 1, we provide a more complete summary, while in appendix B we provide further discussion on the topic.

Topic	Year	Target cipher	ML techniques	Ref.
identification of encryption methods	2006,2010	DES, 3DES, Blowfish, AES, RC5 in CBC	SVM and regression	[16,17]
identification of encryption methods	2012	DES/AES in ECB/CBC	Linear Classifier and SVM	[18]
identification of encryption methods	2018	DES, Blowfish, ARC4, Rijndael, Serpent and Twofish in ECB/CBC, RSA	C4.5, PART, FT, Complement Naive Bayes, MLP and WiSARD	[19]
decryption and distinguishing	2018	DES in ECB/CBC	LSTM and CNN	[20]
ciphertext prediction	2019	DES,Triple-DES	4 or 5 layer MLP	[21,22]
ciphertext prediction	2019	3round-DES, Hitag2	1-6 Layer MLP/Cascade networks	[23]
key recovery	2010,2012	Simplified DES	Levenberg–Marquardt & single MLP	[24,25]
key recovery & understand differential cryptanalysis relation with MLP	2014	Simplified DES	MLP	[26]
key recovery (ASCII key)	2020	S-DES, SIMON32/64, SPECK32/64	3 to 5 layer MLP	[27]
key schedule inversion	2020	PRESENT	3 layer MLP	[28]

Table 1: Summary of the main results regarding machine learning techniques applied to black box cryptanalysis.

The closest related work to this one is [23]. In this work the authors claim to be able to mimic the 1-round DES with accuracy of 99.7% and 2-rounds DES with accuracy of 60% with 2^{17} plaintext/ciphertext pairs. In the same paper, they also analyze the stream cipher Hitag2, being able to mimic the full cipher with 2^{16} input/output pairs, obtaining about 60% accuracy. In this section we analyze this work from the Boolean functions point of view.

Analysis of reduced-round DES. In a reduced 1-round DES not all the bits depend on the same number of inputs. In particular, since DES has a Feistel

structure, the dependencies are different for the bits in the two words, the left and the right one. For the 32 bits of the right word, the dependency is exactly on 1 input bit each, so there should be no problem in learning this word. For the other word, the only non-linearity is given by the S-Box. DES' S-Boxes take 6 input bits, so each bit should depend on a maximum of 7 input bits (the 6 S-Box inputs and the bit itself at the input). Therefore, we think that it is possible to mimic the 1-round DES with neural networks, also reducing the data from 2^{17} to at most $32 \cdot 2^7 = 2^{12}$ chosen inputs. In the case of 2-rounds DES we can apply a similar reasoning from the previous paragraph. The right word at the end of the second DES round will depend on the left word of the output of the first round, so every bit will depend roughly on 7 input bits. For the other word, things become harder: using a similar reasoning with the S-Boxes of DES we can see that every bit of the left word will depend on at most $6 \cdot 7 + 1 = 43$ input bits. In this case, we think that it is possible to mimic the right word, while a lot of data will be required for the left one. Notice that in this case it is possible to reach 75% accuracy with only 2^{12} chosen inputs as follows: 1. Train a neural network to recognize only the right word. Since the depencency is only on the output of the first round, this can be done as described before for 1-round DES. This will get accuracy 100% for this part. 2. For the other word, roughly 2^{48} chosen inputs are necessary, so we assume that this is not feasible and leads to accuracy 50%. 3. The average accuracy of the network will then become 75%.

Analysis of Hitag2. Hitag 2 is a stream cipher based on an LFSR and several Boolean functions. In this case it is not very clear what the authors are doing. From what we understood, they are training the neural network using the "serial" as input and predicting one bit of output, in a fixed-key setting. This is in line with our analysis, since the output bit depends only on 15 bits of the serial number, and so 2^{16} training pairs are more than enough to obtain 60% accuracy.

Other works. In [22,21] the author claim to be able to mimic the full DES and 3DES with 2^{11} and 2^{12} plaintext/ciphertext pairs respectively. We think that, following our previous discussion, these results are unlikely to be reproducible. The same thesis is supported by the authors of [23].

5 Emulating Boolean functions using neural networks

In this section we first describe some experimental results to confirm the theoretical claims we made in section 3 on the minimum number of samples or on the minimum number of neurons (in a single hidden layer MLP) that are needed to obtain accuracy 1 when emulating a Boolean function. Some of these experiments determine the fundamental blocks we used to model 2 rounds of (a reduced version of) AES in section 6. As a side result, we briefly try to correlate the learning rate of a training with some of the main cryptographic properties of a Boolean function in appendix D.

5.1 Experimental results when varying number of samples and neurons

Experiment 1 - Modeling Boolean function depending on a subset of all variables. In this test we investigate Boolean functions which only depend on a subset of all variables. This experiment is motivated by the fact that, in the first rounds, before full diffusion is reached, the output bits of a block cipher usually depend on only some of the input bits. We show that in this case, to reach high accuracy, the needed number of samples grows exponentially in the variables on which the Boolean function actually depends on. Let us recall that we call *noisy* the bits from which the Boolean function does not depend on. In Experiment 2 we will show that the needed number of samples grows exponentially also with the noisy bits.

The experiment works as follows. Pick a random Boolean function of m variables $x_0, \ldots, x_{m-1}$ which only depends on at most m_p of the possible inputs. For example, consider $m = 4, m_p = 2$ and the functions $f_0(x_0, x_1)$, $f_1(x_0, x_1)$, $f_2(x_2, x_3)$, $f_3(x_2, x_3)$. Train an MLP with an input layer of m neurons, a single hidden layer of 2^m neurons and an output layer with a single neuron using t samples.

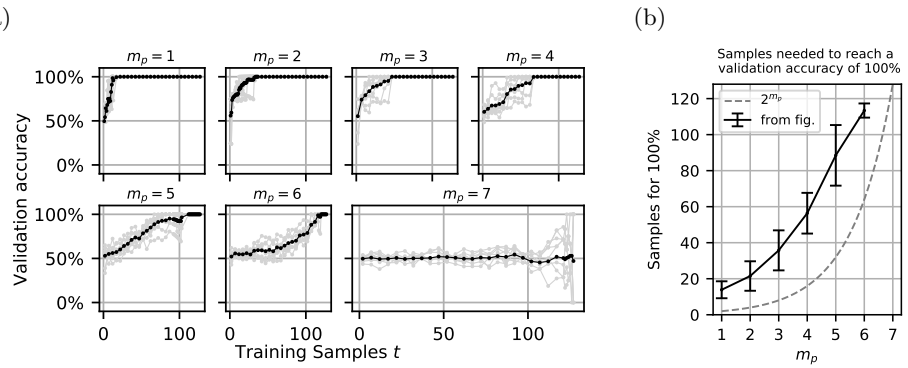

Fig. 2: Results on Experiment 1 for Random Boolean functions of $m = 7$ bits and $m_p = 1, \ldots, m$ dependent variables. Figure (a) shows the block accuracy on the validation dataset for training samples t between $1, \ldots, 2^m - 1$. Each black line shows the mean of ten Random Boolean functions (shown in grey) with $m = 7$ and the indicated m_p. Figure (b) shows the number of samples at which a validation accuracy of 100% has been reached in (a). The number of samples shown are $(13 \pm 4,\ 21 \pm 8,\ 35 \pm 11,\ 56 \pm 11,\ 88 \pm 16,\ 113 \pm 3)$ for the different values of $m_p = 1, \ldots, 6$. For comparison, 2^{m_p} is shown.

The results on this experiment for $m = 7$ ($m_p = 1, \ldots, 7$) are shown in Figure 2. Indeed, for $m_p = 7$ the absolute validation accuracy never reaches 100%, as predicted in Conjecture 1. However, when the number of dependent variables

m_p is smaller, already a fraction of the training samples is sufficient to reach 100% prediction accuracy on an unknown sample.

In particular, for m_p bits, we only need the 2^{m_p} possible values to be presented at least once. So, in principle, 2^{m_p} samples would be enough to reach full accuracy on an unknown sample. In order to estimate how many of the 2^m samples we need (on average) to have the 2^{m_p} values represented, we refer to a modified version of the *coupon collector problem*. If $m - m_p$ is not too small, the expected value for the number of needed samples can be approximated with the classic bound $2^{m_p} \ln(2^{m_p})$ [29]. Using again $m_p = 2$ we have that on average 5.55 samples are enough to have all 4 values for those bits represented. However, as shown in figure Figure 2b more samples are needed.

Experiment 2 - Adding noisy bits to the training. The purpose of this experiment is to show that if we try to model a Boolean function depending on m bits with a neural network taking $m + s$ inputs of which s (the noisy bits) are either fixed to zero or to a random value, it becomes more difficult to obtain a good accuracy, even though for the fixed zero case, accuracy 1 is reached eventually. The experiment works as follows. Pick 1 Boolean function of m variables, add s bits of noise (either fixed to 0 or randomly chosen) and train a neural network with 2^m samples and 2^{m+s} neurons. The results of Experiment 2

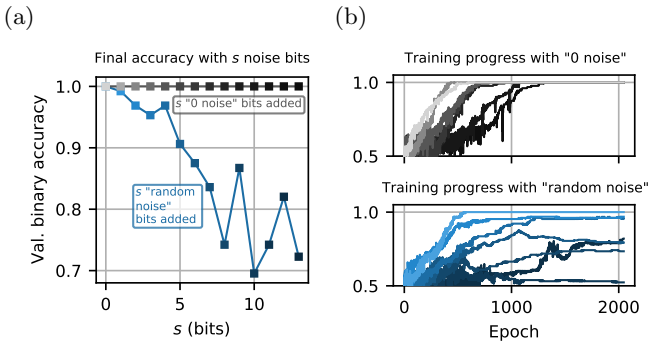

Fig. 3: Results on Experiment 2 for $m = 8$ and $s = 0, \ldots, 13$. Figure (a) shows the final binary accuracy on the validation dataset when the noise bits are either fixed to 0 ("0 noise") or random ("random noise"). Figure (b) shows the validation binary accuracy during training for the final values shown in figure (a). A darker shade corresponds to more noisy bits s.

are shown in Figure 3. We conclude that training becomes harder with increasing s, and that the random noise accentuates this difference.

Experiment 3 - Finding minimum number of samples. The purpose of this experiment is to determine the minimum number of samples for which we reach a high accuracy in the presence of noisy bits, with just one epoch and then with more than one epoch. The experiment works as follows. Pick a random Boolean function of $m + s$ variables $f(x_0, \ldots, x_{m+s-1})$ such that m variables bring information and s variables bring noise. Then find the minimum number

m	n=#Samples	$l = log_2(n)$	l/m
4	25650	14.6	3.65
6	52652	16.7	2.78
8	194385	17.6	2.20
10	2097056	21.0	2.10

m	n=#Samples	$l = log_2(n)$	l/m
4	24883	14.6	3.65
6	36153	15.1	2.52
8	103932	16.7	2.09
10	952149	19.9	1.99

m	n=#Samples	$l = log_2(n)$	l/m	Epochs
4	195	7.6	1.90	336
6	1663	10.7	1.78	151
8	9927	13.3	1.66	103
10	424209	18.7	1.87	78

Table 2: Results for Experiment 3 for one epoch (on the left the results for accuracy=1, in the middle for accuracy$\geq$0.95) and multiple epochs (on the right, with threshold accuracy 0.9, the last column includes 75 epochs of patience, where the training binary accuracy does not improve).

of samples (e.g. with a binary search) for which the neural network reaches an accuracy above the chosen threshold. For the experiment, we fixed $s = 3m$, so that in total we have $4m$ bits of input to the network (this proportion is the same as in 2 rounds of AES-128).

The results are shown in Table 2. From those results, one could estimate that, with just one epoch, $2^{2.1m}$ samples are enough to reach accuracy 1, while 2^{2m} samples are enough to reach at least accuracy 0.95. In the case of more than one epoch, this bound seems to lower towards $2^{1.9m}$. As we explain in section 6, after 2 rounds of AES-128, each output bit is a Boolean function of 32 of the 128 input bits of the cipher. This means that, if our assumption on the growth of the difficulty of the training is correct, then, in order to emulate 2 rounds of AES-128, we need 2^{67} samples to reach accuracy 1 and 2^{64} samples to overcome accuracy 0.95. Since this numbers are too prohibitive for our resources, we will prove our claim to be true for a smaller version of AES (see section 6).

Experiment 4 - Finding the minimum number of neurons. The purpose of this experiment is to determine the minimum number of neurons in the hidden layer which is sufficient to obtain a binary accuracy close to 1. We start picking a random Boolean function of $m + s$ variables $f(x_0, \ldots, x_{m+s-1})$ such that m variables contain information and s variables are noisy bits. As in Experiment 3, we fixed $s = 3m$ and the number of samples and epochs according to Table 2. MLPs with different number of neurons in the hidden layer are trained. The relationship between the number of neurons and the accuracy is shown in Figure 4.

Experiment 5 - Finding the optimal shape of the network. We tried to train networks with increasing number of layers while keeping the same numbers of neurons. We observed no improvements: the reached accuracy is the same of (or even lower than) the networks with a single hidden layer. This was expected, since a single layer neural network with m inputs and 2^m neurons is a universal approximator.

6 Emulating AES using neural networks

In this section we first introduce the internal structure of AES and of a scaled variant. We then use this variant to demonstrate how one can fully model 2 rounds of AES with a limited number of samples.

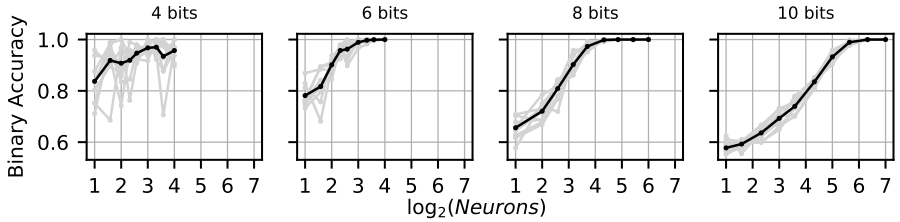

Fig. 4: Training binary accuracy of the neural network from Experiment 4 with a batch size of 100, number of samples and of epochs from Table 2 for different values of $m = 4, 6, 8, 10$.

6.1 AES specifications

In 1997 the National Institute of Standards and Technology (NIST) called for proposals for a new block cipher standard, to be named the *Advanced Encryption Standard* (AES). In October 2000, the Rijndael algorithm, a Belgian block cipher designed by Joan Daemen and Vincent Rijmen [30], was selected as the winner. Nowadays, AES is the most used block cipher.

The AES comes in three different versions that share the same encryption algorithm. At a high level, it can be seen as an alternating key cipher, that is an iterated cipher with the following structure: $E(k, m) = k_d \oplus \pi_d(k_{d-1} \oplus \pi_{d-1}(\dots \pi_1(k_0 \oplus m) \dots))$. The XOR operation $\oplus$ is usually referred to as the `AddRoundKey` operation, where each π_i is defined as the composition of three operations: `SubBytes`, `ShiftRows` and `MixColumns`. For design reasons, π_d omits the `MixColumn` step. Reduced versions of the AES can be considered for experimental purposes, as it was done for example in [31] or [32]. Following a similar approach, in our experiments, we consider a reduced version of the AES where we change the word size and, accordingly to that, the block size. In particular, we consider 4×4 states and 3 bit words. We chose the Sbox of the `SubBytes` operation as the inversion over $\mathbb{F}_2^w$, and an MDS matrix for the `MixColumn` operation [30].

All the operations are computed over $\mathbb{F}_2^w$ were w is the word size in bits. In particular, for 3 bit words, the modulus is the polynomial $x^3 + x + 1$. Like the standard AES, the AES version that we propose reaches full diffusion within 4 rounds. We denote it by AESw3s4.

6.2 AES emulation

Experiment 2 in section 5 is equivalent to predicting a word of a reduced version of AES that performs at most 2 rounds (from the third round, each output bit depends on all the input ones). As noted in the previous section, each output bit of 2 rounds of AES-128 depends on $m = 32$ bits only (1/4 of the total input bits). In the toy AESw3s4, after 2 rounds, each output bit depends on $m = 12$ bits only (again 1/4 of the total input bits). According to Table 2, one needs 2^{2m} samples to be able to emulate the Boolean function defining each output bit

with accuracy of 95%. For AES-128, this means 2^{64}, which is out of reach for our resources. For AESw3s4, only 2^{24} samples are needed. So, we tried to emulate a single output bit of 2 rounds of AESw3s4, using an MLP of 2^{24} neurons fed by 2^{24} samples in the training phase. The experiment was run on a GPU server with 8 Quadro RTX 8000 GPUs, 256 GB RAM and 2 CPUs Intel(R) Xeon(R) Gold 5122 at 3.80 GHz. The test reached a peak of approximately 80 GB of RAM and was terminated after 40 minutes of data generation, 30 minutes of training and 15 minutes of validation. We reached a validation loss of 0.018 and a validation accuracy of 99.6% after 10 epochs.

7 Conclusion

In this work we have shown that to model with high accuracy a random Boolean function one needs to train a neural network with the entire set of all possible inputs of the function. Since the output of any modern block cipher can be represented as a vector of random Boolean functions of n inputs, this means that 2^n samples needs to be used for the training phase, which makes this approach impractical. Nonetheless, there are examples in the literature where this approach was successful, either on full or reduced round ciphers. We explain that when this was possible, it was due to the fact that the output bits of the cipher depend only on a small number of input bits. We exploit this observation to model 2 rounds of (a scaled version of) AES.

References

1. Ronald L Rivest. Cryptography and machine learning. In *International Conference on the Theory and Application of Cryptology*, pages 427–439. Springer, 1991.
2. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 19. The MIT Press, 2017.
3. Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.
4. David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
5. Sepp Hochreiter and J Urgen Schmidhuber. Long Shortterm Memory. *Neural Computation*, 9(8):17351780, 1997.
6. G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, dec 1989.
7. Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.
8. Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 1991.
9. Martin Anthony. Connections between neural networks and boolean functions. *Boolean Methods and Models*, 20, 2005.
10. Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of symmetric neural networks. *Advances in neural information processing systems*, 27:855—-863, 2014.
11. Michael J Kearns. *The computational complexity of machine learning*. MIT press, 1990.
12. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 241–264. ACM, 2019.
13. Bernd Steinbach and Roman Kohut. Neural networks–a model of boolean functions. In *Boolean Problems, Proceedings of the 5th International Workshop on Boolean Problems*, pages 223–240, 2002.
14. Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. *arXiv preprint arXiv:1410.1141*, 2014.
15. Eran Malach and Shai Shalev-Shwartz. Learning boolean circuits with neural networks. *arXiv preprint arXiv:1910.11923*, 2019.
16. Aroor Dinesh Dileep and Chellu Chandra Sekhar. Identification of block ciphers using support vector machines. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 2696–2701. IEEE, 2006.
17. Sammireddy Swapna, AD Dileep, C Chandra Sekhar, and Shri Kant. Block cipher identification using support vector classification and regression. *Journal of Discrete Mathematical Sciences and Cryptography*, 13(4):305–318, 2010.
18. Jung-Wei Chou, Shou-De Lin, and Chen-Mou Cheng. On the effectiveness of using state-of-the-art machine learning techniques to launch cryptographic distinguishing attacks. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, pages 105–110, 2012.
19. Flávio Luis de Mello and José AM Xexéo. Identifying encryption algorithms in ECB and CBC modes using computational intelligence. *J. UCS*, 24(1):25–42, 2018.
20. Linus Lagerhjelm. Extracting information from encrypted data using deep neural networks, 2018.

21. Mohammed M Alani. Neuro-cryptanalysis of des. In *World Congress on Internet Security (WorldCIS-2012)*, pages 23–27. IEEE, 2012.
22. Mohammed M Alani. Neuro-cryptanalysis of DES and Triple-DES. In *International Conference on Neural Information Processing*, pages 637–646. Springer, 2012.
23. Ya Xiao, Qingying Hao, and Danfeng Daphne Yao. Neural cryptanalysis: Metrics, methodology, and applications in CPS ciphers. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8. IEEE, 2019.
24. Khaled M Alallayah, Waiel FA El-Wahed, Mohamed Amin, and Alaa H Alhamami. Attack of against simplified data encryption standard cipher system using neural networks. *Journal of Computer Science*, 6(1):29, 2010.
25. Khaled M Alallayah, Alaa H Alhamami, Waiel AbdElwahed, and Mohamed Amin. Applying neural networks for simplified data encryption standard (sdes) cipher system cryptanalysis. *Int. Arab J. Inf. Technol.*, 9(2):163–169, 2012.
26. Moisés Danziger and Marco Aurélio Amaral Henriques. Improved cryptanalysis combining differential and artificial neural network schemes. In *2014 International Telecommunications Symposium (ITS)*, pages 1–5. IEEE, 2014.
27. Jaewoo So. Deep learning-based cryptanalysis of lightweight block ciphers. *Security and Communication Networks*, 2020, 2020.
28. Manan Pareek, Dr. Girish Mishra, and Varun Kohli. Deep learning based analysis of key scheduling algorithm of present cipher. Cryptology ePrint Archive, Report 2020/981, 2020. `https://eprint.iacr.org/2020/981`.
29. Philippe Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search, 1992.
30. Daemen Joan and Rijmen Vincent. The design of Rijndael: AES-the advanced encryption standard. In *Information Security and Cryptography*. springer, 2002.
31. Carlos Cid, Sean Murphy, and Matthew JB Robshaw. Small scale variants of the aes. In *International Workshop on Fast Software Encryption*, pages 145–162. Springer, 2005.
32. Raphael Chung-Wei Phan. Mini advanced encryption standard (mini-aes): a testbed for cryptanalysis students. *Cryptologia*, 26(4):283–306, 2002.
33. C. Carlet. Boolean functions for cryptography and error correcting codes. *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 257–397, 2010.
34. F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes. I.* North-Holland Publishing Co., Amsterdam, 1977. North-Holland Mathematical Library, Vol. 16.
35. Sean O'Neil and Nicolas Courtois. Reverse-engineered Philips/NXP Hitag2 Cipher, 2008. Available at: `http://fse2008rump.cr.yp.to/00564f75b2f39604dc204d838da01e7a.pdf`.
36. Henryk Plötz and Karsten Nohl. Breaking hitag2. *HAR2009*, 2011, 2009.
37. Nicolas T Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the hitag2 stream cipher. In *International Conference on Information Security*, pages 167–176. Springer, 2009.
38. Petr Štembera and Martin Novotny. Breaking hitag2 with reconfigurable hardware. In *2011 14th Euromicro Conference on Digital System Design*, pages 558–563. IEEE, 2011.
39. Vincent Immler. Breaking hitag 2 revisited. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 126–143. Springer, 2012.

40. Edward F Schaefer. A simplified data encryption standard algorithm. *Cryptologia*, 20(1):77–84, 1996.

Appendix A Preliminaries on Boolean functions

We introduce here, for completeness, the relevant notions concerning Boolean functions. For a complete overview of the topic see [33] or [34].

We denote by $\mathbb{F}_2$ the binary field with two elements. The set $\mathbb{F}_2^n$ is the set of all binary vectors of length n, viewed as an $\mathbb{F}_2$-vector space. A *Boolean function* is a function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$. The set of all Boolean functions from $\mathbb{F}_2^n$ to $\mathbb{F}_2$ will be denoted by $\mathcal{B}_n$.

We assume implicitly to have ordered $\mathbb{F}_2^n$, so that $\mathbb{F}_2^n = \{x_1, \ldots, x_{2^n}\}$. A Boolean function f can be specified by a *truth table* (or *evaluation vector*), which gives the evaluation of f at all x_i's. Once the order on $\mathbb{F}_2^n$ is chosen, i.e. the x_i's are fixed, the truth table of f uniquely identifies f.

A Boolean function $f \in \mathcal{B}_n$ can be expressed in another way, namely as a unique square free polynomial in $\mathbb{F}_2[X] = \mathbb{F}_2[x_1, \ldots, x_n]$, more precisely $f = \sum_{(v_1,\ldots,v_n)\in\mathbb{F}_2^n} b_{(v_1,\ldots,v_n)} x_1^{v_1} \cdots x_n^{v_n}$. This representation is called the *Algebraic Normal Form* (ANF).

There exists a simple divide-and-conquer butterfly algorithm ([33], p. 10) to compute the ANF from the truth-table (or vice-versa) of a Boolean function, which requires $O(n2^n)$ bit sums, while $O(2^n)$ bits must be stored. This algorithm is known as the *fast Möbius transform*.

We now define a set of properties of Boolean functions that are useful in cryptography. In appendix D we study the relation of this properties with the learnability of a Boolean function. We refer to [33] for more details.

The degree of the ANF of a Boolean function f is called the *algebraic degree* of f, denoted by $\deg f$, and it is equal to the maximum of the degrees of the monomials appearing in the ANF. The *correlation immunity* of a Boolean function is a measure of the degree to which its outputs are uncorrelated with some subset of its inputs. More formally, a Boolean function is correlation-immune of order m if every subset of at most m variables in $\{x_1, \ldots, x_n\}$ is statistically independent of the value of $f(x_1, \ldots, x_n)$. The parameter of a Boolean function quantifying its resistance to algebraic attacks is called *algebraic immunity*. More precisely, this is the minimum degree of $g \neq 0$ such that g is an annihilator of f.

The *nonlinearity* of a Boolean function is the distance to the linear functions, i.e. the minimum number of outputs that need to be flipped to obtain the output of a linear function.

Finally, a Boolean function is said to be *resilient* of order m if it is *balanced* (the output is 1 or 0 the same number of times) and correlation immune of order m. The *resiliency order* is the maximum value m such that the function is resilient of order m.

Appendix B Neural networks in black box cryptanalysis: previous results

B.1 Cipher identification

Neural networks can be used to distinguish the output of a cipher from random bit strings or from the output of another cipher, by training the network with

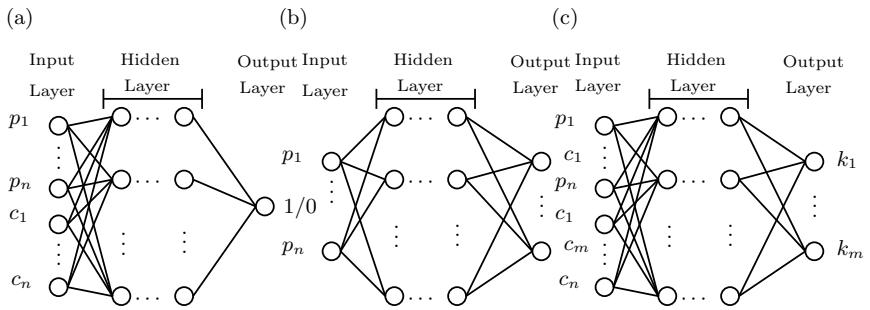

Fig. 5: (a) Generic multilayer perceptron (MLP) architecture to perform a distinguisher attack in known plaintext scenario. The MLP receives n-bit plaintext $p_1, \ldots, p_n$ and ciphertext $c_1, \ldots, c_n$ as input. Each bit serves as input to one neuron, therefore the input layer consists of $2n$ neurons. The output layer consists of a single neuron with two possible outputs, depending on the outcome of the distinguishing attack. (b) Generic multilayer perceptron architecture to perform ciphertext emulation in a known plaintext scenario. (c) Generic multilayer perceptron architecture to map a key recovery attack in the known plaintext scenario. Given plaintext $p_1, \ldots, p_n$/ciphertext $c_1, \ldots, c_n$ pairs as input, each neuron in the output layer predicts one bit of the key $k_1, \ldots, k_m$.

pairs of plaintext-ciphertext obtained from a single secret key (*single secret-key distinguisher*) or from multiple keys (*multiple secret-key distinguisher*). Variations of these attacks might exist in the *related key scenario*, but we are not aware of any work in this direction related to neural networks. The general architecture of neural networks used for distinguisher attacks is shown in Figure 5a.

A direct application of ML to distinguishing the output produced by modern ciphers operating in a reasonably secure mode such as cipher block chaining (CBC) was explored in [18]. The ML distinguisher had no prior information on the cipher structure, and the authors conclude that their technique was not successful in the task of extracting useful information from the ciphertexts when CBC mode was used and not even distinguish them from random data. Better results were obtained in electronic codebook (ECB) mode, as one may easily expect, due to the lack of semantic security (non-randomization) of the mode. The main tools used in the experiment are Linear Classifiers and Support Vector Machine with Gaussian Kernel. To solve the problem of cipher identification, the authors focused on the bag-of-words model for feature extraction and the common classification framework previously used in [16,17], where the extracted features of the input samples are mostly related to the variation in word length. In [18], the considered features are the entropy of the ciphertext, the number of symbols appearing in the ciphertext, 16-bit histograms with 65536 dimensions, the varying length words proposed in [16].

Similar experiments to the one of [18] have also been presented, essentially, with similar results. For example, in [19], the authors consider 8 different plaintext languages, 6 block ciphers (DES, Blowfish, ARC4, Rijndael, Serpent and

Twofish) in ECB and CBC mode and a "CBC"-like variation of RSA, and perform the identification on a higher-performance machine (40 computational nodes, each with a 16-core Opteron 6276 CPU, a NVIDIA Tesla K20 GPU and 32GB of central memory) compared to [18], by means of different classical machine learning classifiers: C4.5, PART, FT, Complement Naive Bayes, MLP and WiSARD. The NIST test suite was applied to the ciphertexts to guarantee the quality of the encryption. The authors conclude that the influence of the idiom in which plaintexts were written is not relevant to identify different encryption. Also, the proposed procedures obtained full identification for almost all of the selected cryptographic algorithms in ECB mode. The most surprising result reported by the author is the identification of algorithms in CBC mode, which showed lower rates than the ECB case, but, according to the authors, the lower rate is "not insignificant", because the quality of identification in CBC mode is still "greater than the probabilistic bid". Moreover, the authors point out that rates increased monotonically, and thus can be increased by intensive computation. The most efficient classifier was Complement Naive Bayes, not only with regard to successful identification, but also in time consumption.

Another recent work is the master thesis of Lagerhjelm [20], in 2018. In this work, long short-term memory networks are used to (unsuccessfully) decipher encrypted text, and convolutional neural network are used to perform classification tasks on encrypted MNIST images. Again, with success when distinguishing the ECB mode, and with no success in the CBC case.

B.2 Cipher emulation

Neural networks can be used to emulate the behaviour of a cipher, by training the network with pairs of plaintext and ciphertext generated from the same key. The general architecture of such networks is shown in Figure 5b. Without knowing the secret key, one could either aim at predicting the ciphertext given a plaintext (encryption emulation), as done, for example, by Xiao et al. in [23], or to predict a plaintext given a ciphertext (decryption emulation), as done, for example, by Alani in [21,22].

In 2012, Alani [21,22] implements a known-plaintext attack based on neural networks, by training a neural network to retrieve plaintext from ciphertext without retrieving the key used in encryption, or, in other words, finding a functionally equivalent decryption function. The author claims to be able to use an average of 211 plaintext-ciphertext pairs to perform cryptanalysis of DES in an average duration of 51 minutes, and an average of only 212 plaintext-ciphertext pairs for Triple-DES in an average duration of 72 minutes. His results, though, could not be reproduced by, for example, Xiao et al. [23], and no source code is provided to reproduce the attack. The adopted network layouts were 4 or 5 layers perceptrons, with different configurations: 128-256-256-128, 128-256-512-256, 128-512-256-256, 128-256-512-128, 128-512-512-128, 64-128-256-512-1024 (Triple-DES), and similar. The average size of data sets used was about 2^{20} plaintext-ciphertext pairs. The training algorithm was the scaled conjugate-gradient. The experiment, implemented in MATLAB, was run on sin-

gle computer with AMD Athlon X2 processor with 1.9 Gigahertz clock frequency and 4 Gigabytes of memory.

In 2019, Xiao et al. [23] try to predict the output of a cipher treating it as a black box using an unknown key. The prediction is performed by training a neural network with plaintext/ciphertext pairs. The error function chosen to correct the weights during the training was mean-squared error. Weights were initialized randomly. The maximum numbers of training cycles (epochs) was set to 10^4. Then, the measure of the strength of a cipher is given by three metrics: cipher match rate, training data, and time complexity. They perform their experiment on reduced-round DES and Hitaj2 [35], a 48-bit key and 48-bit state stream cipher, developed and introduced in late 90's by Philips Semiconductors (currently NXP), primarily used in Radio Frequency Identification (RFID) applications, such as car immobilizers. Note that Hitaj2 has been attacked several times with algebraic attacks using SAT solvers (e.g. [36,37]) or by exhaustive search (e.g. [38,39]).

Xiao et al. test three different networks: a deep and thin fully connected network (MLP with 4 layers of 128 neurons each), a shallow and fat network (MLP with 1 layer of 1000 neurons), and a cascade network (4 layers with 128, 256, 256, 128 neurons). All three networks end with a softmax binary classifier. Their experiments show that the neural network able to perform the most powerful attack varies from cipher to cipher. While a fat and shallow shaped fully connected network is the best to attack the round-reduced DES (up to 2 rounds), a deep-and thin shaped fully connected network works best on Hitag2. Three common activation functions, sigmoid, tanh and relu, are tested, however, only for the shallow-fat network. The authors conclude that the sigmoid function allows a faster training, though all functions eventually reach the same accuracy. Training and testing are performed on a personal laptop (no details provided), so the network used cannot be too large. The training has been performed with up to 2^{30} samples.

B.3 Key recovery attacks

Neural networks can be used to predict the key of a cipher, by training the network with triples of plaintext, ciphertext and key (different from the one that needs to be found). The general architecture of such networks is shown in Figure 5c.

In 2014, Danziger and Henriques [26] successfully mapped the input/output behaviour of the Simplified Data Encryption Standard (S-DES) [40][3], with the use of a single hidden layer perceptron neural network (see Figure 5c). They also showed that the effectiveness of the MLP network depends on the nonlinearity of the internal s-boxes of S-DES. Indeed, the main goal of the authors was to understand the relation between the differential cryptanalysis results and the ones obtained with the neural network. In their experiment, given the plaintext P and ciphertext C, the output layer of the neural network is used to predict the

[3] Notice that S-DES uses 10 bit keys, 8 bit messages, 4 to 2 sboxes, and 2 rounds. This parameters are very far from the real DES.

key K. Thus, for the training of the weights and biases in the neural network, training data of the form (P, C, K) is needed. After training has finished, the neural network was expected to predict a new value of K (not appearing in the training phase) given a new (P, C) pair as input.

Prior works on S-DES include [24,25], where Alallayah et al. propose the use of Levenberg-Marquardt algorithm rather than the Gradient Descent to speed up the training. Besides key recovery, they also use a single layer perceptron network to emulate the behaviour of S-DES, modelling the network with the plaintext as input, and the ciphertext as output. Their results is positive due to the small size of the cipher, and a thorough analysis of the techniques used is lacking.

In 2020, So et al. [27] proposed the use of 3 to 7 layer MLPs (see Figure 5c) to perform a known plaintext key recovery attack on S-DES (8 bit block, 10 bit key, 2 rounds), Simon32/64 (32 bit block, 64 bit key, 32 rounds), and Speck32/64 (32 bit block, 64 bit key, 22 rounds). Besides considering random keys, So et al. additionally restricts keys to be made of ASCII characters. In this second case, the MLP is able to recover keys for all the non-reduced ciphers. It is important to notice that the largest cipher analyzed by So et al. has a key space of 2^{64} keys, which is reduced to $2^{48} = 64^8$ keys when only ASCII keys are considered. The number of hidden layers adopted in this work ranges between 3,5,7, while the number of neurons per layer ranges between 128, 256, 512. In the training phase, So et al. use 5000 epochs and the Adam adaptive moment algorithm as optimization algorithm for the MLP. In comparison to regular gradient descent, Adam is a more sophisticated optimizer which adapts the learning rate and momentum. The training and testing are run on GPU-based server with Nvidia GeForce RTX 2080 Ti and its CPU is Intel Core i9-9900K.

B.4 Key-schedule inversion

As for the emulation of cipher decryption described in subsection B.2, one might try to invert the behavior of the key schedule routine, as done for example by Pareek et al. [28], in 2020. In their work, they considered the key schedule of PRESENT and tried to retrieve the 80-bit key from the last 64-bit round key, using an MLP network with 3 hidden layers of 32, 16, and 8 neurons. Unfortunately, the authors concluded that, using this type of network, the accuracy of predicting the key bits, were not significantly deviating from 0.5.

Appendix C A tiny example

We consider here two parallel Boolean functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$, and suppose we know how two inputs are mapped, i.e. $f_1(00) = 00$, $f_2(01) = 11$. To evaluate the accuracy of an algorithm guessing the output of 10 and 11, one might consider to increase a counter every time

1. the output of the full 2-bits block is guessed correctly. To compute the accuracy, divide the counter by the total number of 2-bit output that have been guessed.

2. the output of the full 2-bit block is guessed correctly for at least 1 bit. To compute the accuracy, divide the counter by the total number of 2-bit output that have been guessed.
3. a single bit is guessed correctly (over all guessed outputs). To compute the accuracy, divide the counter by the product of bits per block (2) and the total number of 2-bit output that have been guessed.

As an example, let us suppose that the correct missing values are mapped to $f_1(10) = 01$, $f_2(11) = 11$. Let us also suppose that an algorithm $\mathcal{A}$ made the following guess $10 \mapsto 00$, $11 \mapsto 10$. According to the first metric the accuracy of $\mathcal{A}$ is 0. According to the second metric the accuracy of $\mathcal{A}$ is 1. According to the third metric, the accuracy of $\mathcal{A}$ is $3/4$.

Note that if we have to guess two 2-bit Boolean functions mapping $00 \mapsto 00$, $01 \mapsto 11$, $10 \mapsto 01$, then we can correctly guess where the value 11 will be mapped to with probability $1/4$. On the other hand, if we know that the two Boolean functions have to form a permutation over the set $\{00, 01, 10, 11\}$, then we only have the option $11 \mapsto 10$. In general, if there are r missing values for a set of m' m-bit Boolean functions, and we know they have to form a permutation ($m' = m$), we can guess correctly with probability $1/r!$. If the m' m-bit Boolean function does *not* necessarily form a permutation, then we can guess correctly with probability $1/(2^{rm'})$, which is much lower than $1/r!$. In the case of a block cipher, we also know that not all permutations are possible, but only the ones indexed by the n-bits keys, which are 2^n.

Appendix D Emulating Boolean functions with different cryptographic properties

In this section, we want to determine if there exist a correlation between the learnability of a Boolean function and some of its most relevant cryptographic properties, namely: algebraic degree, algebraic immunity, correlation immunity, nonlinearity and resiliency order (see appendix A or [33] for definitions).

We randomly picked ten Boolean functions, in $m = 10$ variables, for each algebraic degree from $1, \ldots, 9$ (i.e. 90 Boolean functions in total). A neural network was trained to predict the output of these functions. In Figure 6a it is shown how the neural network parameters affect the accuracy of the predictions (for the case of algebraic degree property), while Figure 6b shows the network performance during the training. In both graphs, we take, for each value of the algebraic degree, the average of the accuracy and the loss over the ten Boolean functions considered.

In particular, we notice two facts. The first one is that we need the full dataset in order to be able to predict the outcome of the Boolean functions. The second one is the similarity of the training progress for all algebraic degrees (with a slight irregularity in linear functions) in Figure 6b, which points out that the algebraic degree is not causing major differences in the learnability of the Boolean functions.

The panels in figure Figure 6c show the training progress for the algebraic immunity, the correlation immunity, the nonlinearity and the resiliency order.

While for the algebraic immunity and nonlinearity no major differences in the training progress are visible, we notice that for correlation immunity and resiliency order there are some differences in the training progress. The results on correlation immunity are in line with the work from Malach et al. [15], but a detailed investigation is beyond the scope of this work and is left for future research.

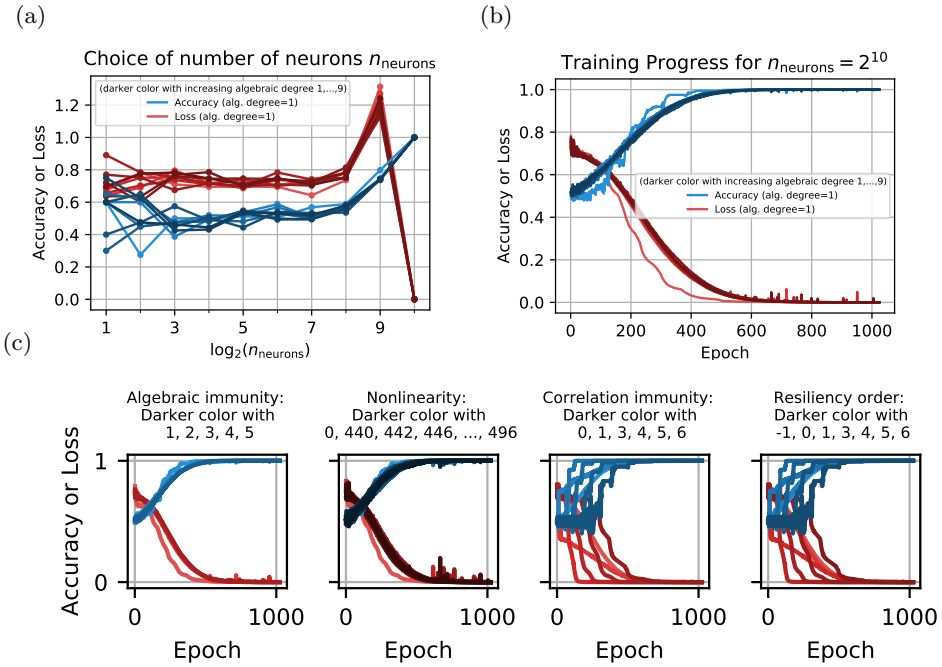

Fig. 6: Binary accuracy (blue) and binary crossentropy loss (red) of an MLP learning Boolean functions of varying algebraic degree. The left hand side figure (a) shows the final accuracy and loss values obtained on the validation dataset for different configurations $e = 1, \ldots, 10$. In detail the number of neurons in the hidden layer of the MLP was varied ($2^e = 2^1, \ldots, 2^{10}$), as well as the number of samples (2^e) and number of training epochs (2^e). The right hand side figure (b) shows the training progress of a neural network with 1024 neurons, 1024 samples and 1024 epochs. Figure (c) in the lower panel shows the training progress of a neural network with 1024 neurons, 1024 samples and 1024 epochs for various other considered properties of Boolean functions.