Deep Neural Cryptography

David Gerault[1], Anna Hambitzer[1], Eyal Ronen[2], and Adi Shamir✉[3]

[1] Cryptography Research Center, Technology Innovation Institute
{david.gerault,anna.hambitzer}@tii.ae
[2] Tel Aviv University
eyalronen@tauex.tau.ac.il
[3] Weizmann Institute
adi.shamir@weizmann.ac.il

Abstract. The wide adoption of deep neural networks (DNNs) raises the question of how can we equip them with a desired cryptographic functionality (e.g, to decrypt an encrypted input, to verify that this input is authorized, or to embed a secret backdoor which alters the DNN's normal operation). The problem is that cryptographic primitives are typically designed to run on digital computers that use Boolean gates to map sequences of bits to sequences of bits, whereas DNNs are a special type of analog computer that uses linear mappings and ReLUs to map vectors of real numbers to vectors of real numbers. This discrepancy between the discrete and continuous computational models raises two fundamental questions: How can we implement standard cryptographic primitives as DNNs, and whether DNN implementations of secure cryptosystems remain secure in the new setting, in which an attacker can ask the DNN to process a message whose "bits" are arbitrary real numbers. In this paper we lay the foundations of this new area of research, defining the meaning of correctness and security for implementations of cryptographic primitives as ReLU-based DNNs. We then show that the natural implementations of block ciphers as DNNs can be broken in linear time by using such nonstandard inputs (this attack was experimentally tested in the case of full round AES-128, and had 100% success rate). Finally, we develop a new method for implementing any desired cryptographic functionality as a standard ReLU-based DNN in a provably secure and correct way. Our novel protective technique has very low overhead (a constant number of additional layers and a linear number of additional neurons), and is completely practical.

Keywords: Deep learning, cryptography, cryptanalysis, deep neural cryptography, secure implementation.

1 Introduction

Two highly active areas of research within Computer Science are deep learning and cryptography, but so far very little had been done to combine them (unlike the situation at the intersection of deep learning and cyber security, where there had been a lot of fruitful integration). One of the main obstacles is the fact that

the two areas are based on completely different models of computation: most cryptographic schemes are mappings over the discrete domain of bits and are implemented as a composition of Boolean gates, whereas DNNs are mappings over the continuous domain of real numbers and are implemented as a composition of linear mappings and nonlinear activation functions such as Rectified Linear Units (ReLUs). In this paper, we study the central problem at the intersection of deep learning and cryptography, namely whether one can implement standard cryptographic primitives in deep neural networks while maintaining their security.

To clarify the research question, suppose that we want to implement the Advanced Encryption Standard (AES) block cipher as a DNN. Due to the high expressive power of such networks, there are several natural ways to do it in such a way that for any standard plaintext (whose bits are zeroes and ones) provided as input to the DNN, the output of the DNN will be the correct standard ciphertext (which also consists of zeroes and ones). However, due to the domain extension from bits to real valued inputs, such a DNN will also provide some output when it is given the nonstandard input whose first "bit" is 0.3, whose second "bit" is -7, whose third "bit" is π, etc. Due to the continuity of linear mappings and ReLUs, such nonstandard outputs will be some kind of nonlinear but continuous interpolation between the standard outputs for nearby standard inputs. While the legitimate user is only interested in binary inputs and outputs, an adversary can try to use nonstandard real valued plaintext bits in his attack in order to extract the secret cryptographic key from the DNN. This is reminiscent of side channel attacks in which legitimate users only provide 0 or 5 volts as inputs to a smart card, while hackers can try to "glitch" it by providing unusual voltages between 0 and 5 volts, which do not correspond to valid digital inputs. Similarly, when we implement cryptography on a quantum computer, an attacker can feed it with entangled superpositions of qubits instead of bits. Thousands of papers published over the last few decades have shown that these two domain extensions can drastically change the security of cryptosystems; in this paper we show the same for the newly proposed field of deep neural cryptography.

It is important to note that DNNs cannot be trained via gradient descent to perform typical cryptographic operations such as secret key encryption, since by design the input/output relation is hard to generalize from a small number of training examples. In addition, trained DNNs usually provide only an approximation of the desired functionality, whereas we want to have a perfectly correct 0/1 outputs for 0/1 inputs. The only realistic way to implement such a mapping is to synthesize the DNN as a composition of its basic operations. In such an implementation, the secret key bits are either stored in a special inaccessible register whose values are fed into the DNN, or provided as some additional inputs to the DNN which cannot be seen or manipulated by the attacker. Note that in this case, the attacker cannot try to break the cryptosystem by choosing real-valued key bits that are not 0s or 1s, and cannot use related key attacks in which he tries to flip some key bits in order to check its effect on the ciphertext.

Bridging Digital and Analog Security Models. One of the main motivations for our research was the recent paper "Planting Undetectable Backdoors in Machine Learning Models" by Goldwasser, Kim, Vaikuntanathan and Zamir [9]. In their paper, the authors used standard cryptographic primitives in order to hide a backdoor in the DNN. However, since such primitives are only defined over the discrete domain of $\{0,1\}^n$, they stated that all their DNN inputs should be restricted to such values. Under this restriction, they could use the following lemma which was attributed to Minsky and Papert [17]:

"Lemma 3.2: Given a Boolean circuit C of constant fan-in and depth d, there exists a multilayer perceptron N of depth d computing the same function."

However, typical DNNs (such as image classifiers) are essentially analog computers, which accept floating point real values as inputs, multiply them with real valued weights, and use the continuous ReLU as the activation function. So while all the claims in their paper are technically correct, they "missed all the fun" of analyzing other potential values. This is similar to claiming that a smart card is secure because it correctly implements the digital AES, while completely ignoring the dangers posed by analog side channel attacks.

As we show in this paper, the security of cryptosystems over 0/1 inputs *does not imply* that their domain extension to real-valued inputs (via a perfectly correct DNN implementation) is also secure. In fact, we show that all the natural implementations of AES as a DNN can be broken in linear time by analyzing the effect of changing each input "bit" around its initial value x by a tiny amount ϵ into $x+\epsilon$ and $x-\epsilon$.

One possible countermeasure implementers can try to use is to "sanitize" the inputs in order to prevent potentially harmful values from being processed by the DNN. Unfortunately, there is no ReLU-based DNN gadget which can map 0 to 0, 1 to 1, and any other input to either 0 or 1, since any such function must have at least one discontinuity point whereas any ReLU-based DNN is always continuous. However, we can partially sanitize the inputs by adding one extra layer in front of the DNN which applies to each input separately the function $\texttt{ClippedReLU}(x) = \texttt{ReLU}(x) - \texttt{ReLU}(x-1)$. This function clips any negative x to 0, clips any x larger than 1 to 1, and leaves any input x between 0 and 1 unchanged. However, as we show in this paper such partial sanitization of all the inputs and outputs of the DNN does not suffice to protect the cryptographic functionality, since it can still be attacked with a different (and slightly more complicated) attack. Note that in certain applications (such as the insertion of a hidden backdoor into a standard DNN which distinguishes cats from dogs, as described in [9]), the mere inclusion of a non-DNN based sanitization process will by itself be a telltale sign that a digital backdoor may have been inserted into the analog DNN.

In the last part of the paper we finally solve the fundamental security problem of deep neural cryptography by describing a new protective technique which uses only standard ReLU-based DNN's and yields a provably secure implementation of any desired cryptographic functionality. We achieve this by forcing the DNN to continuously interpolate between its standard outputs (for standard binary

inputs) in a way which cannot possibly reveal any new information about the secret key.

In addition to bridging the conceptual gap between digital and analog security models, we have three additional motivations for this work. **i) Foundational Motivation: Introducing rigorous Security Notions for DNNs.** Formal security notions for DNNs remain mostly unexplored, and new AI applications introduce novel attack surfaces. Our new framework makes it possible to use standard cryptographic tools and proof techniques in such applications.
ii) Practical Motivation: Efficiently securing Deep Neural Cryptography. We show that our provably secure DNN-based implementations of standard cryptographic primitives can be highly practical. As shown in appendix C, our secure neural implementation of AES adds only a tiny overhead when used as preprocessing layers in contemporary DNN inference. This shows that deep neural cryptography is not purely of theoretical interest but could indeed be feasible for real-world deployment in systems that combine machine learning with data confidentiality.
iii) Forward-Looking Motivation: Enabling New Potential Applications. There is currently a major effort (especially in China, see [3]) to develop purely analog photonic chips for AI applications. Embedding cryptographic primitives within such neural networks may enable new privacy-preserving ML workflows. For example, in federated learning, one could train on strongly encrypted data while keeping the cryptographic layer within the model stack. Due to space limitations, we only consider one such application in appendix C, but note that our secure constructions make many similar directions more realistic.

Contributions The main contributions of this paper are:

1. Formalizing the notions of a correct and secure implementation of a cryptographic functionality in a DNN (see section 3 and section 9).
2. Demonstrating that for some cryptographic task which is defined over bits but implemented as a DNN, there is a provably exponential gap between the difficulty of solving the problem when the adversary can use only bits and when he can use real numbers as inputs (see section 5).
3. Defining the notion of natural implementations of cryptographic functionalities as DNNs, which compose the DNN versions of their basic Boolean operations (see section 4 and section 6).
4. Showing that such natural implementations of cryptosystems are completely insecure in the sense that an adversary who can feed the DNN with real numbers as plaintext "bits" can extract all their embedded secret key bits in linear time (see section 7).
5. Demonstrating that just clipping the inputs to the range $[0, 1]$ does not solve this insecurity problem (see section 7.3).
6. Experimentally verifying our key recovery attacks on natural implementations of AES-128 with 100% success rate (see section 8).
7. Developing a new way of implementing any key-based cryptographic functionality as a DNN which is provably secure in the sense that any information

about the secret key bits which can be obtained by using real-valued inputs can also be obtained by using only zeroes and ones as inputs (see section 9).

2 Related Work

Encrypting Analog Data

Cryptography has very strong roots in the discrete domain, and the design of cryptographic primitives that process continuous data is uncommon. However, such primitives flourished for a time during WW2 due to the need for *secure speech* systems, to protect the confidentiality of telephonic communications. One notable example is the SIGSALY [21] system from Bell Labs, which swapped ten frequency bands and six amplitude bands based on a secret key.

After this period, cryptosystems that processed real-valued data have not received a lot of attention, as most of them can be shown to be insecure (in particular, they suffer from the fact that in any continuous mapping, nearby plaintexts are always encrypted into nearby ciphertexts). It is important to realize that we do not face the same problem in this paper, since our goal is different: in our case, the legitimate user of the scheme only wants to encrypt standard plaintexts which are strings of zeroes and ones, but he wants to use an analog type of computational device which can also accept real-valued inputs prepared by an adversary.

Boolean Functions, Cryptography and Neural Networks

The idea of implementing arbitrary Boolean logic through a neural network, therefore showing the universality of this computational model, has been central to the field since its inception, when Pitts and Mc Culloch [16] suggested that biological neurons, through their threshold activation mechanism, could compute Boolean functions. The first practical construction of an artificial neuron, the perceptron of Rosenblatt [20], was criticized for its inability to learn non-linearly separable Boolean functions (such as XOR) by Minsky and Papert [17], possibly triggering the first so-called "AI winter". The study of the representation power of multilayer perceptron culminated with the universal approximation theorem established by Cybenko [6] and Hornik et al. [11], stating that they can approximate any continuous function on a compact domain.

While it has been known for a long time that neural networks can be used to build any Boolean circuits, for instance with `ReLU` activations in [18], the implementation of cryptographic primitives into neural network has received very little attention. Recently, Goldwasser, Kim, Vaikuntanathan and Zamir [9] proposed a backdoor construction for neural networks based on a cryptographic signature algorithm, implemented as described in [17], with a non-continuous step activation function. This idea of adding cryptographic backdoors to neural networks was also extended to large language models in [8] [12].

Finally, some attempts based on training, rather than building, DNNs have shown little success [4], and essentially concluded that the required amount of data for training is exponential in the state size.

Neural Networks for Cryptography

The problem of building cryptography using neural networks has been more popular. For instance, in [13], Kanter *et al.* propose to iteratively synchronize two randomly and independently initialized neural networks through their answers to public queries until convergence, and to derive a shared cryptographic key from their states at the end of the procedure. However, this key exchange mechanism was quickly shown to be insecure [14], and subsequent attempts have also been shown to be vulnerable.

Abadi and Anderson [2] extended this idea to adversarial neural cryptography, where the neural networks are given a shared key, and synchronize to learn a secure encryption scheme. This work uses three neural networks: Alice encrypts its plaintext input with the shared key, Bob attempts to invert the encryption with the key, and Eve without it. The three neural networks are trained in turns with different goals: Bob and Eve's loss depends on their ability to reconstruct the plaintext, while Alice's loss function aims to maximize Bob's score while minimizing Eve's advantage. In [5], the authors generalize the framework to enhance Eve with chosen plaintext abilities, and use a dedicated architecture to permit the neural networks to learn the XOR function. The encryption algorithms produced by such techniques have so far been very simple, and their security relies heavily on the ability to use the one time pad through their unlimited key material.

Cryptography for Neural Networks

The design of techniques to enable either learning or inference on encrypted data is a very active research field [15], where dedicated neural networks are used to support the use of binary inputs; this differs from our goal in this work, which is to build a secure encryption functionality into a generic, continuous neural network, where adversarial real-valued queries are a strong attack vector.

3 Preliminaries

3.1 Basic DNN Definitions and Notations

Definition 1 (ReLU Activation Function). *The* ReLU *(Rectified Linear Unit) activation function, which maps negative values to 0, is defined as*

$$\texttt{ReLU}(x) \;=\; \max\{0,\, x\}.$$

Definition 2 (Neuron). *A* neuron *is a function η determined by a weight (column) vector w, a bias b, and an activation function f, which computes*

$$\eta(x) = f(x \cdot w + b).$$

Definition 3 (Deep Neural Network (DNN)). *A* Deep Neural Network *is a function $f : \mathcal{R}^{d_0} \to \mathcal{R}^{d_{r+1}}$ built as a composition of neurons arranged into* layers. *Layer l is defined by a weight matrix W^l, bias vector b^l and activation functions list σ^l, such that neuron j of layer l has weights $W^l_{\cdot,j}$, bias b^l_j and activation σ_j.*

Throughout this paper, we focus on ReLU-based neural networks, which only have linear mappings and ReLUs; we sometimes loosely use the term DNN to refer to such neural networks.

3.2 The Threat Model

Let $\mathcal{D}$ be a DNN-based implementation of some binary keyed cryptographic primitive $\mathcal{B}$ (e.g., AES encryption, HMAC, RSA signing, etc).

The binary implementation $\mathcal{B}$ has a binary key space $\mathcal{K} \subseteq \{0,1\}^\lambda$, a binary input space $\mathcal{P}_{\texttt{Bin}} \subset \{0,1\}^*$, and a binary output space $\mathcal{C}_{\texttt{Bin}} \subset \{0,1\}^*$, such that:

$$\mathcal{B} : \mathcal{K}, \mathcal{P}_{\texttt{Bin}} \to \mathcal{C}_{\texttt{Bin}}.$$

The DNN-based implementation of the same cryptographic primitive $\mathcal{D}$ has (the same) key space $\mathcal{K} \subseteq \{0,1\}^\lambda$, an infinite precision real valued input space $\mathcal{P}_{\mathcal{R}} \subset \mathcal{R}^*$, and an output space $\mathcal{C}_{\mathcal{R}} \subset \mathcal{R}^*$, such that:

$$\mathcal{O} : \mathcal{K}, \mathcal{P}_{\mathcal{R}} \to \mathcal{C}_{\mathcal{R}}.$$

For the sake of brevity, we will omit the key from our notation, unless it is specifically required.

Throughout this paper, we will assume that our DNN-based implementation $\mathcal{D}$ is "correct", i.e., it agrees with the binary implementation $\mathcal{B}$ on all binary inputs. More formally:

Definition 4 (Correctness of DNN-based implementation). *For any DNN-based implementation $\mathcal{D}$ of a binary cryptographic primitive $\mathcal{B}$, $\mathcal{D}$ is correct if*

$$\forall \boldsymbol{k} \in \mathcal{K} \wedge \forall p \in \mathcal{P}_{\texttt{Bin}} : \mathcal{B}_{\boldsymbol{k}}(p) = \mathcal{D}_{\boldsymbol{k}}(p).$$

Our threat model assumes an adaptive attacker with oracle access to $\mathcal{D}$. The attacker can ask for the output value of $\mathcal{D}$ on any chosen input values, where each input "bit" is a real number. Our adaptive attacker can choose their next input value based on the results of the previous queries.

The main question we ask ourselves is, what is the relative security of our DNN-based implementation $\mathcal{D}$ that can receive real-valued inputs, compared to the binary implementation $\mathcal{B}$ that can only receive binary inputs?

We will show that in many cases, such an adaptive attacker can exploit queries with non-binary input values to extract the keys of "natural" DNN-based implementations. However, in section 9, we will introduce some novel DNN-based gadgets, and show how to use them in order to construct a blackbox transformation so that the resultant DNN provably satisfies our security definition.

4 Natural Implementations of Small Boolean Functions as A DNN

In this section, we describe the simplest way to implement any Boolean function over a small number of bits as a ReLU-based neural network, which we refer to in the rest of the paper as its *natural implementation* [4]. We first illustrate the technique through the example of the XOR of two bits, before describing the general case.

4.1 Natural Implementation of XOR

The XOR function of two inputs x_1, x_2 can be implemented in a simple one layer DNN with two neurons which compute the following two intermediate values C_1, C_2:

$$C_1 = \texttt{ReLU}(x_2 - x_1)$$
$$C_2 = \texttt{ReLU}(x_1 - x_2)$$

The value $y = C_1 + C_2$ (which is the absolute value of the difference between the two inputs, and thus measures the real-valued distance between them) produces the desired XOR value for any combination of $\{0, 1\}$ values of x_1 and x_2, and continuously extends this definition to $\mathcal{R}^2$. Notice that when we embed this XOR in a larger circuit, there is no need to spend a second DNN layer to compute this addition operation - any subsequent neuron that needs the value y will simply incorporate the computation of $C_1 + C_2$ into the linear mapping of its inputs.

4.2 Natural Implementations of Other Boolean Functions

The natural implementations we present in this subsection generalize this technique to arbitrary Boolean functions over a small number k of input bits through a *corner summation* construction. In particular, we will use this method in section 6 to naturally implement the SBox of the AES block cipher (which maps 8 input bits to 8 output bits) as a small ReLU-based DNN.

Consider any Boolean function $f : \{0, 1\}^k \to \{0, 1\}$. We again add only one layer, which contains one neuron for each binary input $b = (b_1 \dots b_k)$ such that $f(b) = 1$. We call any such vertex b of the k-dimensional Boolean hypercube an *active corner*, and denote the set of all the active corners by $\mathcal{C}$. The output of the implementation of f is the sum of the outputs of all these intermediate values. Note that this layer contains at most 2^k neurons, which is not too large whenever k is a small constant such as $k = 8$.

[4] We briefly discuss a more generic, but less natural variant in section A

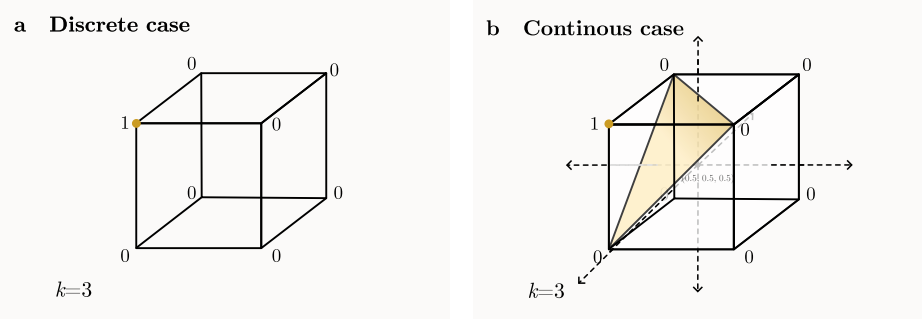


Fig. 1: A three dimensional corner function. **a.** The Boolean hypercube with one active corner. **b.** The zero hyperplane of this corner function.

We now describe the linear mapping of the neuron associated with a particular active corner $b = (b_1 \dots b_k)$ of f. We want this linear mapping to output the value 1 at the associated corner, and negative or zero values at all the other corners of the Boolean hypercube (which will be turned into 0's by the neuron's ReLU). We achieve this by orienting the zero hyperplane of the linear mapping in such a way that it splits the Boolean hypercube into two half-spaces so that the positive side of the linear mapping will contain only b, and the negative side of the linear mapping will contain all the other $2^k - 1$ corners, as depicted in Figure 1b. The formal definition of this corner function is the following:

Definition 5 (Corner Function). *: The corner function associated with the active corner $b = (b_1 \dots b_k) \in \{0,1\}^k$ is:*

$$\texttt{corner}_{b_1 \dots b_k}(x) \;=\; \texttt{ReLU}\left(\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} (1 - x_i) - k + 1\right)$$

It is easy to verify that this function outputs the value one at corner b, zero at any other corner of the Boolean hypercube, and a value in $[0;1]$ for any other point inside the hypercube. In addition, if $\texttt{corner}_b(x) > 0$ for some $x \in [0;1]^n$, then any corner function $\texttt{corner}_{b'}(x)$ such that b' differs from b in more than a single coordinate outputs 0. On the other hand, a unique corner adjacent to b may be activated as well; in that case, the two outputs sum to 1.

When the Boolean function f has more than one active corner, we formally define its implementation as:

Definition 6 (Sum of Corners). *Let $f : \{0,1\}^k \to \{0,1\}$ be a Boolean function, with active corners $\mathcal{C} = \{b = (b_1 \dots b_k) : f(b) = 1\}$. The continuous generalization of f is the sum of the corner functions of all its active corners:*

$$\Sigma_{\mathcal{C}}(x) = \left(\sum_{b \in \mathcal{C}} \texttt{corner}_{b1 \dots b_k}(x)\right)$$

Note that the implementation of XOR described in the previous subsection is a special case of this general construction in two dimensions with two active corners at $(0, 1)$ and $(1, 0)$.

Finally, we define the natural implementation of a *vectorial Boolean function* $f : \{0,1\}^k \to \{0,1\}^m$ as the concatenation of the m sums, placed side by side in the DNN. This construction is formalized in Algorithm 1, which describes how to implement a vectorial Boolean function $f : \{0,1\}^k \to \{0,1\}^m$ as a neural network with k input neurons, $\sum_{o=1}^{m} |\mathcal{C}_o|$ neurons in the hidden layer, and m output neurons which implement the m sums of corners.

Algorithm 1 `NN(f)`

Input: A Boolean function $f : \{0,1\}^k \to \{0,1\}^m$.
Output: The natural implementation $\mathtt{NN}_f$.

`NN` $\leftarrow k$ input neurons ▷ Build input layer of `NN`
`/* Build first hidden layer of NN with` $\sum_{o=1}^{m} |\mathcal{C}_o|$ `neurons: */`
for $o \in [1; m]$ **do**
  $\mathcal{C}_o \leftarrow \{x : f(x)_o = 1\}$ ▷ Collect corners.
  for $x \in \mathcal{C}_o$ **do**
    $w \leftarrow (w_1, \ldots, w_k)$ ▷ Initialize weights and biases.
    $b \leftarrow 1 - k + \sum_{i:x_i=0} (1)$
    for $x_i \in x$ **do**
      if $x_i = 0$ **then**
        $w_i \leftarrow -1$
      else
        $w_i \leftarrow 1$
    `NN` $\leftarrow (w, b,$ `ReLU`$)$ ▷ Add neuron to layer, with ReLU activation
`/* Build output layer of NN: */`
for $o \in [1; m]$ **do**
  $w = [1] * |\{x : f(x)_o = 1\}|$ ▷ All weights have value 1.
  $b = 0$
  `NN` $\leftarrow (w, b)$ ▷ Add $\Sigma_{\mathcal{C}_o}$ to layer
return network `NN`

These natural implementations of Boolean functions over a small number of bits (which can also be used for Boolean functions over a large number of bits when they have only a small number of active corners) can be composed into arbitrarily complex structures in order to implement any Boolean circuit as a DNN. Note that the sum of corner functions and the parallel evaluation of the m outputs can again be absorbed into the linear mapping of the next layer, and thus require only one (rather than two) additional layer in the DNN.

5 A Provably Exponential Gap Between The Complexity of a Problem With Binary and Real Queries

In this section, we show the existence of a simple search problem which is defined over binary inputs and can be naturally implemented with a blackbox DNN with a single neuron, which can be solved exponentially faster when real valued queries are allowed.

The problem we consider is the well known problem of unstructured search, in which the Boolean hypercube has a single hidden corner $b \in \{0,1\}^k = (b_1, \ldots, b_k)$ at which $f(b) = 1$, and at all the other corners the value of f is zero. We are given blackbox access to the natural single neuron implementation of f as a DNN, and our goal is to find the hidden corner b with the smallest possible number of queries under two scenarios: When all the queries must be binary, and when real-valued queries are allowed. We will show that in the first scenario there is a provable lower bound of $\Omega(2^k)$ on the complexity of the problem, while in the second scenario there is a simple $O(k)$ algorithm for solving it. Incidentally, the same unstructured search problem was shown by Grover to be solvable quadratically faster under a different type of domain extension (from bits to qubits).

In the Boolean version of the problem, the expected number of queries needed to find b is exponential in the input size k, since the only information which can be gained by querying the $2^k - 1$ nonactive corners of the Boolean hypercube and obtaining the answer 0 is that they are not the correct hidden corner.

Consider now the second scenario, in which we are allowed to use arbitrary queries from $\mathcal{R}^k$ when interacting with the natural single neuron implementation of f. We can start from the center of the cube (at point $x = (0.5, \ldots, 0.5)$, which is always on the negative side of the hyperplane) and try to find the positive half space of this neuron by going sufficiently far in the positive or negative directions of each one of the coordinates x_i separately, as depicted by the six dashed lines in Figure 1. It is easy to verify that for the natural definition of the corner function from Definition 5, it suffices to move a distance of $\frac{k}{2}$ from the center of the cube in order to obtain a positive output, provided that we move in the correct (positive or negative) direction of the i-th coordinate. Consequently, we can deduce that $b_i = 1$ if we obtain a positive output when we increase x_i by $\frac{k}{2}$, and that $b_i = 0$ if we still get a value of 0 after increasing x_i. We can thus solve the k dimensional hidden corner problem in this scenario with exactly k queries.

6 A Natural Implementation of The AES

In this section we describe the natural implementation of the Advanced Encryption Standard (AES) as a DNN denoted by $\mathtt{NN}_{\mathtt{AES}}$, which uses sums of corner functions to implement all its basic operations. The concrete details of our pytorch implementation are further discussed in section B.

The AES is a block cipher which encrypts 128-bit plaintexts with 128, 192 or 256 bit keys. The 128-bit key variant, denoted AES-128, applies 10 iterations

of a round function to the state, represented as a 4×4 matrix of bytes. This round function is composed of four operations:

- AddRoundKey (ARK): An XOR with the round key
- SubBytes (SB): A non-linear bytewise substitution layer
- ShiftRows (SR): A circular rotation of the state rows
- MixColumns (MC): A matrix multiplication applied to the columns of the state; it is omitted in the last round.

The round keys are build from the master key through the key schedule algorithm, which we assume is run ahead of time to embed the keys directly into the neural network when it is built. Note that the first round key $\mathtt{k}^1$ is equal to the master key $\mathtt{k}$, so anyone who can find the first round key can easily derive from it all the other round keys.

The high level description of our implementation is given in Algorithm 2. All the Boolean functions of the implementation are defined using $\mathtt{NN}(\cdot)$.

Algorithm 2 $\mathtt{NN}_{\mathtt{AES}_\mathtt{k}}(p)$

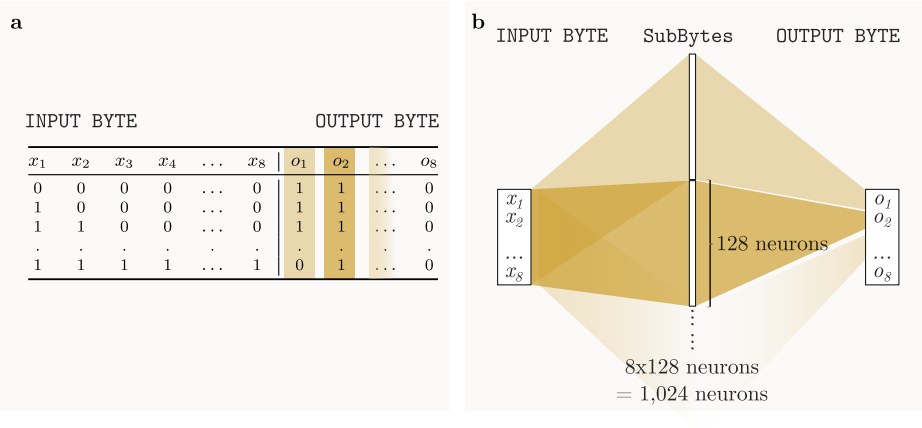
Input: A 128-bit plaintext p and a 128-bit key $\mathtt{k}$.
Output: 128-bit ciphertext.

$(\mathtt{k}^1, \ldots, \mathtt{k}^{11}) \leftarrow$ *AES key schedule*$(\mathtt{k})$ ▷ Derive round keys from AES key schedule
$x \leftarrow \mathtt{NN}_{\mathtt{ARK}}(p, \mathtt{k}^1)$
for $i \in [1, \ldots, 9]$ **do**
    $x \leftarrow \mathtt{NN}_{\mathtt{SB}}(x)$
    $x \leftarrow \mathtt{NN}_{\mathtt{SRMC}}(x)$
    $x \leftarrow \mathtt{NN}_{\mathtt{ARK}}(x, \mathtt{k}^{i+1})$
$x \leftarrow \mathtt{NN}_{\mathtt{SB}}(x)$
$x \leftarrow \mathtt{NN}_{\mathtt{SR}}(x)$
$x \leftarrow \mathtt{NN}_{\mathtt{ARK}}(x, \mathtt{k}^{11})$
return x

Bitwise XOR: $\mathtt{NN}_{\mathtt{XOR}}$ Bitwise XOR is implemented as $\mathtt{NN}((x_1, x_2) \rightarrow x_1 \oplus x_2))$. In our implementation, we often need to perform XORs of bytes or 128-bit words, which corresponds to the concatenation of the neurons of multiple $\mathtt{NN}_{\mathtt{XOR}}$ networks; we denote these with the shorthand $\mathtt{NN}_{\mathtt{XOR}8}$ and $\mathtt{NN}_{\mathtt{XOR}128}$. Similarly, in the MC operation, we use the notation $\mathtt{NN}_{\mathtt{XOR}8}(\cdot, \cdot, \cdot, \cdot)$ for the XOR of 4 bytes, obtained by combining 8 4-way XORs $\mathtt{NN}((x_1, x_2, x_3, x_4) \rightarrow x_1 \oplus x_2 \oplus x_3 \oplus x_4)$.

AddRoundKey $\mathtt{NN}_{\mathtt{ARK}}$**.** The network $\mathtt{NN}_{\mathtt{ARK}}$ applies the natural implementation of XOR between the 128 bits of the state p and a 128-bit round key $\mathtt{k}$, using $\mathtt{NN}_{\mathtt{XOR}128}(p, \mathtt{k})$.

SubBytes $\texttt{NN}_{\texttt{SB}}$. The SubBytes operation applies a bijective SBox to each byte of the 128-bit state. In our implementation, we concatenate the natural implementations of the Boolean functions of each of the 8 output bits, $\texttt{NN}((x_1 \dots x_8) \rightarrow \texttt{SB}(x)_i)$. Each of the resulting eight networks has 128 neurons which represent the corner functions of the 128 active corners in the 8-dimensional Boolean cube representing one output bit of the SB operation.



x_1	x_2	x_3	x_4	...	x_8	o_1	o_2	...	o_8
0	0	0	0	...	0	1	1	...	0
1	0	0	0	...	0	1	1	...	0
1	1	0	0	...	0	1	1	...	0
.	.	.	.	.	.	.	.	.	.
1	1	1	1	...	1	0	1	...	0

Fig. 2: **a.** Truth table representation of the SubBytes operation that maps eight input bits $x_1, \dots, x_8$ to a single output bit $o_1, \dots, o_8$. **b.** Neural network implementation $\texttt{NN}_{\texttt{SBOX}}$ of the 8 outputs in the truth table.

In total, the corresponding network has $8 \times 128 = 1,024$ neurons in the hidden layer that performs a single SB operation, as visualized in Figure 2b.

ShiftRows and MixColumns $\texttt{NN}_{\texttt{SRMC}}$. The ShiftRows transformation applies a left rotation by i bytes for each row i, which is implemented by applying the appropriate permutation when connecting the outputs of $\texttt{NN}_{\texttt{SBOX}}$ to the inputs of the next operation.

The MixColumns operation multiplies the state by a 4×4 matrix, with coefficients 1, 2 and 3, corresponding to multiplications in the Rijndael finite field. These multiplications are implemented using lookup tables [1] $\texttt{MUL}_t$ defined over 8-bit inputs, such that $\texttt{MUL}_t[x]$ returns the multiplication of input x by the constant t. In our natural implementation, we concatenate the natural implementations of the corresponding Boolean functions to build $\texttt{NN}_{\texttt{MUL2}}$ as $\texttt{NN}((x_1 \dots x_8) \rightarrow \texttt{MUL}_2(x)_i)$, and $\texttt{NN}_{\texttt{MUL3}}$ as $\texttt{NN}((x_1 \dots x_8) \rightarrow \texttt{MUL}_3(x)_i)$.

The application of MixColumns on column j of the state, $\texttt{NN}_{\texttt{MC}j}$ (Algorithm 3), combines these intermediate values using $\texttt{NN}_{\texttt{XOR8}}$.

The 128-bit state after MixColumns is build as the concatenation of the 4 32-bit column outputs.

Algorithm 3 $\texttt{NN}_{\texttt{MC}_j}(x)$

Input: A vector $x = (x_1 \dots x_{32})$
Output: The output of the MixColumns operation on x
1: $a_0, b_0, c_0, d_0 \leftarrow (x_1 \dots x_8), (x_9 \dots x_{16}), (x_{17} \dots x_{24}), (x_{25} \dots x_{38})$
2: $a_1 \leftarrow \texttt{NN}_{\texttt{XOR8}}(\texttt{NN}_{\texttt{MUL2}}(a_0), c_0, d_0, \texttt{NN}_{\texttt{MUL3}}(b_0))$
3: $b_1 \leftarrow \texttt{NN}_{\texttt{XOR8}}(\texttt{NN}_{\texttt{MUL2}}(b_0), a_0, d_0, \texttt{NN}_{\texttt{MUL3}}(c_0))$
4: $c_1 \leftarrow \texttt{NN}_{\texttt{XOR8}}(\texttt{NN}_{\texttt{MUL2}}(c_0), a_0, b_0, \texttt{NN}_{\texttt{MUL3}}(d_0))$
5: $d_1 \leftarrow \texttt{NN}_{\texttt{XOR8}}(\texttt{NN}_{\texttt{MUL2}}(d_0), b_0, c_0, \texttt{NN}_{\texttt{MUL3}}(a_0))$
6: return (a_1, b_1, c_1, d_1)

7 Attacking This Natural DNN Implementation

7.1 Attack Overview

In this section, we demonstrate the existence of a simple key recovery attack on the natural implementation of any block cipher (such as AES) in which the encryption operation starts with an XOR between the plaintext and the key. This attack can be easily extended to other common structures (e.g., when a plaintext byte and a key byte are added rather than XOR'ed). Our attack can deal with block ciphers with arbitrarily many rounds with arbitrarily complicated round functions, and runs in linear time (as a function of the number of bits in the first round key).

Consider the initial operation of XOR'ing one key bit $\texttt{k}_i$ and one plaintext bit x_i, as represented in Figure 3. The binary key bit is fixed and cannot be seen or modified by the attacker, while the plaintext value can be chosen by the attacker. Note that in this figure, the attacker can move only horizontally, and does not know whether he is at the top or the bottom side of the square since he does not know $\texttt{k}_i$.

In the binary input case, the attacker is restricted to setting x_i to either 0 or 1, and flipping x_i flips the output of the XOR. This can be potentially used in standard (discrete) differential attacks, but no such attack is expected to exist for strong cryptosystems such as full AES-128. On the other hand, we show that in natural DNN implementations of this XOR, we can use a continuous version of differential cryptanalysis by comparing the ciphertexts which are produced by tiny horizontal movements of x_i to $x_i \pm \epsilon$.

Our first attack requires such movements to be bidirectional, which results in some x_i values that are outside of $[0, 1]$. In this case, the output of the XOR will change symmetrically at two corners of the cube and asymmetrically at the two other corners. With high probability, this will remain true as the changes avalanche throughout the remaining rounds of AES, and will enable us to recover each key bit separately. Our second attack lifts this requirement, and demonstrates that partial sanitization of the inputs which forces all of them to remain in $[0, 1]$ is not sufficient to counter key recovery. In this case, we can no longer apply the symmetry test since we can move by ϵ only in one direction. The new idea here is to test whether simultaneous changes in the 8 bits entering a partic-

ular S-box in the first round of AES are blocked by its sum of corner functions, which can only happen when the output of the S-box is eight zeroes. Since such blocked changes in the first round leads to unchanged ciphertexts (after arbitrarily many additional rounds), we can easily identify such cases, and thus recover the eight key bits entering this S-box in the first round.

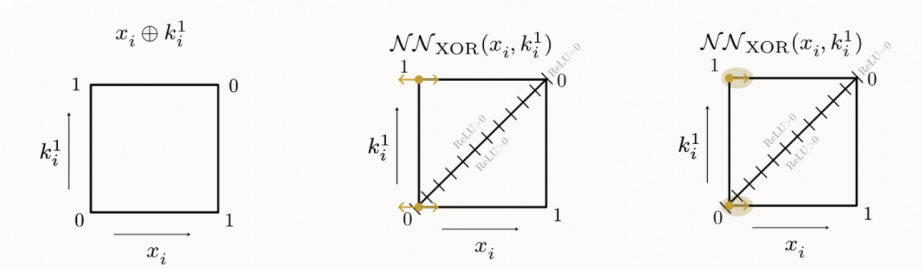


Fig. 3: Our attacks on natural implementations of the XOR. **a.** Binary implementation of the XOR $x_i \oplus \mathtt{k}_i^1$. **b.** A natural implementation of the XOR $\mathtt{NN}_{\mathtt{XOR}}(x_i, \mathtt{k}_i)$. **c.** A partially sanitized version of the natural implementation, in which plaintext values must remain in the range $[0, 1]$.

Our two attack strategies can be implemented by a single procedure, which is invoked with either one or eight as the number of attacked key bits per iteration (denoted by `nbits`), and with two different ways to generate plaintext pairs for which a collision is expected (denoted by `GenPairs`). The attack algorithm `KeyRecovery` is described in Algorithm 4, and its output is the recovered key k'.

The attacks start from a random initial plaintext, and iterate over all target key positions (bits or bytes). For each position i, the `GenPairs` algorithm generates $2^{\mathtt{nbits}}$ pairs of plaintexts, such that the pair in position v encrypts to identical ciphertexts if the target key bit(s) have value v. By encrypting these $2^{\mathtt{nbits}}$ pairs and comparing the resulting ciphertexts, we obtain a list of candidate key values for the targeted bits. If a single candidate is obtained, then the key position is successfully recovered; otherwise, an unlucky cancellation situation had happened, and the position remains unrecovered. The procedure is repeated with a new base plaintext until all positions have been recovered.

7.2 Attack on Natural Implementations with Unsanitized Inputs

This attack exploits the symmetry of the output of the XOR around the diagonal. As noted in section 4.1, $\mathtt{NN}_{\mathtt{XOR}}(x_i, \mathtt{k}_i) = \mathtt{ReLU}(x_i - \mathtt{k}_i) + \mathtt{ReLU}(\mathtt{k}_i - x_i)$ is equal to the absolute value $|\mathtt{k}_i - x_i|$. Therefore, the two inputs $x_i = \mathtt{k}_i - \epsilon, x_i' = \mathtt{k}_i + \epsilon$ produce the same output, $|\epsilon|$, whereas $x_i = 1 - \mathtt{k}_i - \epsilon$ and $x_i' = 1 - \mathtt{k}_i + \epsilon$ produce different outputs for all $\epsilon > 0$.

Algorithm 4 $\texttt{KeyRecovery}(\texttt{NN}_{\texttt{AES}}, \texttt{GenPairs}, \texttt{nbits})$

Input: A handle to a natural implementation of the AES, $\texttt{NN}_{\texttt{AES}_\texttt{k}}(\cdot)$; a pairs generation algorithm $\texttt{GenPairs}$for a variation parameter ϵ returning $2^{\texttt{nbits}}$ plaintext pairs, each corresponding to a candidate value for $\texttt{nbits}$ consecutive key bits.
Output: Candidate $\texttt{k}'$ for key $\texttt{k}$

$\texttt{remainingPositions} \leftarrow \{1, \cdots, \frac{128}{\texttt{nbits}}\}$
while $\texttt{remainingPositions} \neq \emptyset$ **do**
  Select random base plaintext p
  for $i \in \texttt{remainingPositions}$ **do** ▷ For each unknown key position
    $\texttt{candidates} \leftarrow \emptyset$
    $(x^0, x'^0) \dots (x^{2^{nbits}}, x'^{2^{nbits}}) \leftarrow \texttt{GenPairs}(p, i)$
    for Candidate key value $v \in \{0 \dots 2^{\texttt{nbits}} - 1\}$ **do**
      $C \leftarrow \texttt{NN}_{\texttt{AES}}(x^v)$
      $C' \leftarrow \texttt{NN}_{\texttt{AES}}(x'^v)$
      if $C = C'$ **then**
        Add v to $\texttt{candidates}$
    if $\texttt{candidates}$ contains a single value v **then**
      $k'_{i \dots i+\texttt{nbits}} \leftarrow v$
      Remove i from $\texttt{remainingPositions}$
return k'

The $\texttt{GenPairs}^{sym}$ pairs generation algorithm, described in Algorithm 5 builds one pair that is expected to collide after XOR with a key bit $\texttt{k}_i$ equal to 0, and another one that is expected to collide when key bit $\texttt{k}_i$ is equal to 1.

Algorithm 5 $\texttt{GenPairs}^{sym}(p, i, \epsilon)$

$(x^0, x'^0) \leftarrow ((p|p_i = -\epsilon), (p|p_i = \epsilon))$
$(x^1, x'^1) \leftarrow ((p|p_i = 1 - \epsilon), (p|p_i = 1 + \epsilon))$
return $(x^0, x'^0), (x^1, x'^1)$

7.3 Attack on Natural Implementations with Partially Sanitized Inputs

The previous attack depends on the symmetric behaviour of the natural implementation of the bitwise XOR operator for carefully chosen inputs. In particular, it requires querying the encryption of plaintexts p where an input $p_i \in \{0, 1\}$ is changed to both $p_i - \epsilon$ and $p_i + \epsilon$. A natural countermeasure to thwart this attack is to partially sanitize all the inputs and outputs of the encryption function to eliminate unsafe values which are outside $[0, 1]$. This operation can be performed by using the single layer DNN gadget:

$$\texttt{ClippedReLU}(x) = \texttt{ReLU}(x) - \texttt{ReLU}(x - 1)$$

We now describe our second attack which does not rely on symmetry and can thus be applied even when the inputs and outputs are partially sanitized. Instead of targeting the initial XOR of key and plaintext bits, the new attack targets each SBox operation in the first round, and recovers one key byte at a time. It relies on the observation that small changes around the preimage of 00000000 in the SBox of AES (which is 01010010), are completely blocked since for all the 8 output bits, the corresponding corner in the 8-dimensional Boolean cube is not active (for any other SBox input, at least one output bit has the value 1, and is thus an active corner whose corner function will pass the input changes to the outputs). This will enable us to identify cases in which the input to a particular SBox in the first round of AES was 01010010, and thus to recover the 8 key bits which were XOR'ed to the input bits in this round.

The attack in this case has the following structure: For each one of the 256 key byte candidates v, we build the corresponding candidate plaintext byte that goes to 0 after the corresponding SBox in the first layer, $x^v_{i\ldots i+8} = v \oplus 01010010$, and $x'^v_{i\ldots i+8}$ in which all bits are moved by ϵ in the allowed direction (i.e., the direction which will leave them in the range $[0, 1]$). The candidate for which the two corresponding ciphertexts are equal is our key guess. This procedure is described in Algorithm 6.

Algorithm 6 $\texttt{GenPairs}^{clip}(p, i, \epsilon)$

for $v \in \{0 \ldots 255\}$ **do** ▷ For each candidate byte value
  $x^v = (p | p_{i\ldots i+8} = v \oplus 82)$
  $x'^v = (x^v | x^v_j = \text{if } x^v_j = 0 \text{ then } x^v_j + \epsilon \text{ else } x^v_j - \epsilon), j \in \{i \ldots i+8\})$
return $(x^0, x'^0), \ldots, (x^{255}, x'^{255})$

8 Experimental Results

We ran these attacks on the natural implementation of AES-128, and successfully retrieved the entire 128-bit key in all our experiments. The attack using Algorithm 5 for pair generation recovers one key bit at a time, and detects a bit recovery failure when both candidate pairs result in identical ciphertexts; such cases are very rare (they occur with approximate probability 0.003), and all key recoveries were concluded successfully when using at most two additional random plaintexts. Furthermore, the failures can be completely eliminated by adjusting the change parameter ϵ. The results are summarized in Table 1.

8.1 Attacking the Natural Implementation of AES

We ran the attack described in Algorithm 5 on the natural implementation of AES described in section 6. The attack was run for 1000 random keys, which were all successfully recovered, 687 (68.7%) of which on the first try. In the

	ϵ	#K	$n_p = 1$	$n_p = 2$	$n_p = 3$	$n_p = 4$	Failures
$\texttt{GenPairs}^{sym}$	0.1	1000	687	310	3	0	386
	0.01	1000	1000	0	0	0	0
$\texttt{GenPairs}^{clip}$	0.1	1000	1000	0	0	0	0

Table 1: The results of our key recovery attacks on full-round natural implementations of AES, listing, among the total number of attacked keys #K, how many were recovered using one, two, three or four base plaintexts, and the total number of bit failures.

remaining cases, a Fail was detected in one or more recovered key bits, so that 310 (31%) keys required a second base plaintext, and three (0.3%) required a third plaintext. These correspond to a total of 386 bit recovery failures, out of 128000 key bits recovered; all these failures were identified during the attack.

8.2 Failures Analysis

In the attack using PredictKeybitSymm, bit recovery failures were detected for a small number of target key bits.

After our first series of attacks, we reviewed the failure data to extract the base plaintexts, keys, and bit positions corresponding to these failures. We then encrypted the pairs step by step to identify the operation where an unexpected difference cancellation occurred. We observed that all the failure cancellations occurred during the Sbox operation in various rounds.

More specifically, the failures can be traced to internal values approaching 0.5 in the state bits where the difference propagates (and therefore, going to 0 after corner operations); by using a smaller $\epsilon = 0.01$, we can keep the internal state values closer to 0 and 1, which effectively eliminated all the cancellations.

8.3 Attacking Natural AES with Partially Sanitized Inputs

We ran the attack described in Algorithm 6 on a natural implementation of the AES using the Clipped ReLU sanitization. The attack was run for 1000 random keys, which were all successfully recovered on the first try with $\epsilon = 0.1$.

9 Defenses and Security Proof

In this section we describe a generic blackbox secure transformation, which uses only linear and `ReLU` functions; it takes an arbitrary DNN-based implementation $\mathcal{D}$ of some cryptographic primitive $\mathcal{B}$ which uses a secret key, and yields a provably secure implementation $\mathcal{D}_S$ of the same primitive. We start by informally describing our security definition. Then, we will describe our transformation and prove its security. Informally, our transformation achieves the following correctness and security guarantees:

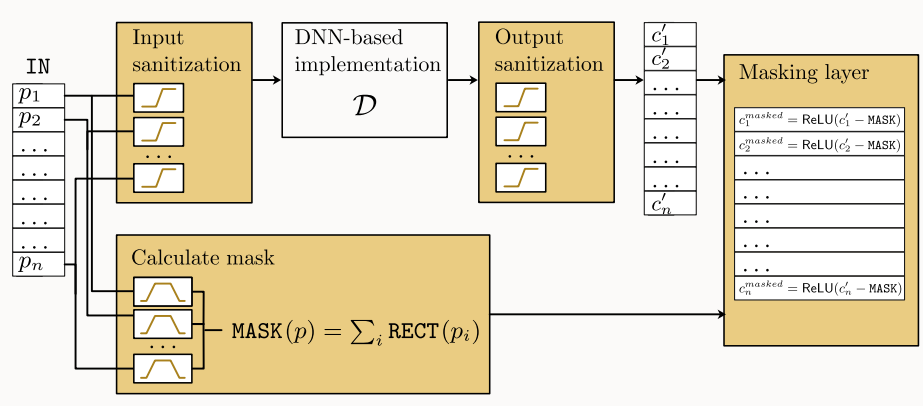


Fig. 4: The overall structure of our secure transformation from $\mathcal{D}$ to $\mathcal{D}_S$

1. **Correctness** — On any binary input (i.e., where all input values are '0' or '1'), the secure DNN-based implementation $\mathcal{D}_S$ will output the same values as the original DNN-based implementation $\mathcal{D}$ (assuming that $\mathcal{D}$ outputs binary outputs on binary inputs).
2. **Security** — Queries to $\mathcal{D}_S$ which use arbitrary real numbers as inputs do not leak any information about the secret key that is not already leaked via binary valued queries to the original $\mathcal{D}$.

Assuming that the original $\mathcal{D}$ behaves correctly on all binary inputs (i.e., returns the same output values as $\mathcal{B}$), this will result in a secure implementation, in which no polynomial time adversary can exploit the ability to use real numbers as inputs to leak information about the secret key, regardless of the specifics of $\mathcal{D}$. Note that $\mathcal{D}_S$ is only as secure as the original cryptographic primitive $\mathcal{B}$, and any possible attack on $\mathcal{B}$ will also apply to our new implementation.

9.1 Blackbox Secure Transformation

Figure 4 shows the overall structure of our transformation, which is comprised of two main parts:

1. Partial sanitization of all inputs and outputs, which uses an approximation of a step function to map the real input and output values to a range of 0 to 1. The resulting mapping partitions the domain of real numbers into "safe" input values, which will always be mapped to binary values of '0' or '1', and a small leftover range of "unsafe" values that will be mapped to real numbers between 0 and 1. The partial sanitization of the outputs ensures that even when the input values are "unsafe", all the output values will be in the range $[0, 1]$.
2. Outputs masking layer, which zeroizes all of the output values if any of the input values are "unsafe". This layer uses an approximation of a rectangular

function that receives the value 1 if the input value is in the "unsafe" range but is 0 if the input value is binary (either '0' or '1'). By summing the rectangular functions over all input values, we get a "masking" value, which is larger than or equal to 1 if even one of the input values is "unsafe" but is 0 if all input values are binary. The masking layer subtracts the masking value from each output value and then applies a `ReLU` function on the result. As we will show, if any of the input values are in the "unsafe" range, all outputs will be 0. If all the input values are binary, the output will be the expected result. Finally, for other input values that are neither unsafe nor binary, the output will be a smooth interpolation between the expected result and zero whose form does not depend on the secret key.

We will now describe the two components in detail, and explain the `STEP` and `RECT` functions that we use.

9.2 Inputs and outputs "sanitization"

Recall that the previously shown DNN-based cryptographic implementations were correct when using only binary values of '0' and '1'. However, an attacker could exploit the ability to use arbitrary real numbers as input to leak information about the key. If we were able to map all possible input values into either '0' or '1', this would prevent all such attacks. One way to do it, is using an ideal step function, that return '0' on all inputs that are smaller than 0.5, and '1' for all inputs equal or larger than 0.5.

$$\texttt{STEP}_{\texttt{IDEAL}}(x) = \begin{cases} 1 & \text{if } x \geq 0.5, \\ 0 & \text{if } x < 0.5 \end{cases}$$

We would like to add an "input sanitization" layer of ideal step functions before our DNN-based implementation, where each input value is mapped to either zero or one before it is used by the original DNN-based implementation. Unfortunately, such an ideal step function cannot be implemented by a `ReLU`-based DNN, since it is not a continuous function.

Approximate Step Function Although we cannot implement an ideal step function, we can implement the following approximate step function using only two `ReLU`s:

$$\texttt{STEP}_{\epsilon}(x) = \begin{cases} 1 & \text{if } x \geq 0.5 + \epsilon/2, \\ 0 & \text{if } x \leq 0.5 - \epsilon/2 \\ \epsilon^{-1} \cdot (x - (0.5 - \epsilon/2)) & \text{if } 0.5 - \epsilon/2 < x < 0.5 + \epsilon/2 \end{cases}$$

We can set ϵ to any value which is strictly larger than 0 and strictly smaller than 1, but for the sake of brevity we will use from now on the particular constant

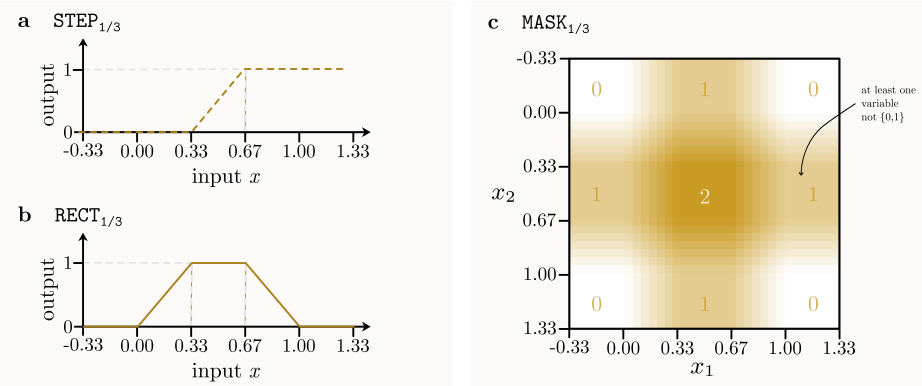


Fig. 5: Visualization of the $\texttt{STEP}_{1/3}$ and $\texttt{RECT}_{1/3}$ functions used in our sanitization layers and the resulting masking value $\texttt{MASK}_{1/3}$ for two-dimensional inputs.

$\epsilon = 1/3$. The function $\texttt{STEP}_{1/3}$, shown in Figure 5a, can be implemented using $\texttt{ReLU}$s as follows:

$$\texttt{STEP}_{1/3}(x) = 3 \cdot (\texttt{ReLU}(x - 1/3) - \texttt{ReLU}(x - 2/3))$$

As this approximate step function is realizable by a $\texttt{ReLU}$-based DNN, we can use it in an "input sanitization" layer. Note that it is different from the $\texttt{ClippedReLU}$ sanitization function we used before (which was the identity in the range $[0, 1]$), since we have to use a narrower range of unsafe inputs in order to leave room for the other elements in our construction to operate.

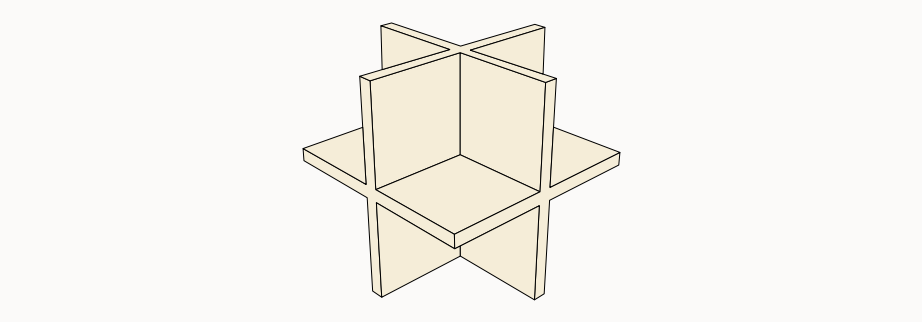


Fig. 6: Visualization of the (truncated) three dimensional P_{unsafe} input domain

We can now define a "safe" input domain P_{safe}, such that for every input $p \in P_{safe}$, every input value p_i is either smaller than $1/3$ (which will be mapped to '0') or is larger than $2/3$ (which will be mapped to '1'). Intuitively, the new input sanitization layer $\texttt{STEP}_{1/3}$ "rounds" all inputs from the "safe" input domain

to '0' and '1', and thus prevents the attacker from exploiting such values in his attack. However, we don't have any guarantees about inputs from the complementary P_{unsafe} input domain. The shape of this P_{unsafe} input domain in three dimensions is described in Figure 6.

To further limit the attacker's ability to exploit values from the P_{unsafe} input domain, we will also add an "output sanitization" layer, where each output value is also passed through $\texttt{STEP}_{1/3}$. [5] This layer has no effect when using inputs from the "safe" domain (as they are all mapped to '0' or '1', and thus create only outputs values of '0' or '1'). However, we get the additional guarantee that even for "unsafe" inputs, all outputs will be limited to the range of $[0, 1]$. We will use this guarantee in our output masking layer that we will describe in the next subsection.

To summarize, we can define the sanitization layer as follows: For input p with input values p_i, we define the sanitized input p' with values ${p'}_i = \texttt{STEP}(p_i)$, the output c with output values c_i, as the output of our DNN-based implementation $\mathcal{D}$ on the sanitized input p' such that $c = \mathcal{D}(p')$, and the sanitized output c' with values $c'_i = \texttt{STEP}(c_i)$.

9.3 Outputs Masking Layer

Although our sanitization layer significantly restricts the attacker's capabilities, we have shown in section 7.3 that attackers can still exploit "unsafe" input values which are not mapped to 0 to 1 to leak information about the key. The second part of our transformation aims to protect against such attackers. This is done by ensuring that if at least one of the inputs p_i to the DNN is "unsafe" (i.e., in the range $1/3 < p_i < 2/3$), then all the output values will be set to zero. The masking layer calculates a masking value which is a function of the input values, that is at least 1 whenever at least one of the inputs is "unsafe", but is zero whenever all of the input values are binary. This masking value is subtracted from each sanitized output value, and then a `ReLU` is applied to the result.

Approximate `RECT`: To calculate our masking value, we can use a rectangular function `RECT` that has the value 1 on the "unsafe" range (between 1/3 and 2/3) and 0 otherwise. Unfortunately, as in the case of the ideal step function, the ideal rectangular function cannot be implemented by a `ReLU`-based DNN as it is not a continuous function. Although we cannot implement such a rectangular function, we can implement the following approximate function $\texttt{RECT}_{1/3}$:

$$\texttt{RECT}_{1/3}(x) = \begin{cases} 0 & \text{if } x \leq 0, \\ 3 \cdot x & \text{if } 0 < x < 1/3 \\ 1 & \text{if } 1/3 \leq x \leq 2/3, \\ 3 \cdot (1 - x) & \text{if } 2/3 < x < 1 \\ 0 & \text{if } x \geq 1, \end{cases}$$

[5] While our corner-based implementations does not require this final sanitization (see section 4.2), it is needed to protect arbitrary implementations.

Figure 5b shows the approximate RECT function $\texttt{RECT}_{1/3}$ which can be implemented using four ReLUs as follows:

$$\begin{aligned}\texttt{RECT}_{1/3}(x) = 3 \cdot (&\texttt{ReLU}(x) - \texttt{ReLU}(x - 1/3) \\ &- \texttt{ReLU}(x - 2/3) + \texttt{ReLU}(x - 1))\end{aligned}$$

Masking With Approximate RECT: For every input p, we define the following masking value $\texttt{MASK}_{1/3}$ as:

$$\texttt{MASK}_{1/3}(p) = \sum_i \texttt{RECT}_{1/3}(p_i)$$

This new masking values partitions P_{safe} into two subdomains. The first domain is the domain P_0 where $\texttt{MASK}_{1/3}(p) = 0$. By definition we get that:

$$p \in P_0 \texttt{ if } \forall p_i \in p, p_i \leq 0 \vee p_i \geq 1$$

The second domain $P_{safe} \setminus P_0$ is the part of the domain which is still "safe", but its masking value is not zero:

$$p \in P_{safe} \setminus P_0 \texttt{ if } p \in P_{safe} \wedge \exists p_i \in p \texttt{ s.t. } 0 < p_i \leq 1/3 \vee 2/3 \geq p_i < 1$$

Finally, we can describe our masking values as:

$$\texttt{MASK}(p) = \begin{cases} 0 & \forall p \in P_0 \\ > 0 & \forall p \in P_{safe} \setminus P_0 \\ \geq 1 & \forall p \in P_{unsafe} \end{cases} \tag{1}$$

Figure 5b shows the masking value for the two dimensional case. Note that in this case the value of the mask on all unsafe inputs is between 1 and 2, it is 0 for all $p \in P_0$ and it is some non zero value for all $p \in P_{safe} \setminus P_0$.

We can now use our masking value in the masking layer to zeroize the output values for all "unsafe" inputs:

$$c_i^{masked} = \texttt{ReLU}(c'_i - \texttt{MASK}(p)) \tag{2}$$

Assuming that the implementation $\mathcal{D}$ is correct on binary inputs, we know that:

$$\forall p \in P_{safe} : p' \in \{0,1\}^* \wedge c' = c = \mathcal{D}(p') \in \{0,1\}^* \tag{3}$$

As $\forall c_i, 0 \geq \texttt{STEP}(c_i) \geq 1$, from equation 1, 2, and 3 we get that:

$$c_i^{masked} = \begin{cases} c_i & \forall p \in P_0 \\ c_i^{masked} & \forall p \in P_{safe} \setminus P_0 \\ 0 & \forall p \in P_{unsafe} \end{cases}$$

Where the value of c_i^{masked} when $p \in P_{safe} \setminus P_0$ is a smooth interpolation between the binary value of c_i and 0.

Algorithm 7 Secure Blackbox Transformation $\mathcal{D}_S$

Input: A possibly insecure DNN-based implementation of a keyed cryptographic primitive $\mathcal{D}$, a key $\mathtt{k}$ and input p.
Output: output c.

1: $p' \leftarrow \mathtt{STEP}_{1/3}(p)$ ▷ Sanitization layer, applied on each input value
2: $\mathtt{MASK}_{1/3} \leftarrow \Sigma_{p_i \in p}(\mathtt{RECT}_{1/3}(p_i))$ ▷ Calculating masking value
3: $c \leftarrow \mathcal{D}(\mathtt{k}, p^{step})$
4: $c' \leftarrow \mathtt{STEP}_{1/3}(c)$ ▷ Sanitization layer, applied on each output value
5: $c^{masked} \leftarrow \mathtt{ReLU}(c' - \mathtt{MASK}_{1/3})$ ▷ Masking layer, applied on each output value
6: **return** c^{masked}

Secure Transformation Overhead: We now show that our secure transformation is highly practical, adding only a small additive complexity to the original (insecure) implementation of the cryptographic functionality. Our transformation requires adding only three layers to the total depth of the DNN. One layer is for the input sanitization, one layer is for the output sanitization, and one layer is for the masking layer. For each input value, we need two `ReLU`s for calculating the `STEP` function in the input sanitization layer and 4 `ReLU`s for the `RECT` function for calculating the mask value. For each output value, we need 2 `ReLU`s for calculating the `STEP` function in the output sanitization layer, and one `ReLU` to calculate the masked output value. Taking our AES DNN-based implementation as an example, adding our secure transformation results in an overhead that is negligible compared to the cost of the original implementation.

9.4 Secure Implementation Correctness

Our full secure transformation is described in Algorithm 7. We start by analyzing the correctness of our secure transformation $\mathcal{D}_S$. We note that any binary input is not affected by our step function, and results in a mask value of 0:

$$\forall p \in \{0,1\}^* : \ p' = \mathtt{STEP}_{1/3}(p) = p \ \wedge \ \mathtt{MASK}_{1/3}(p) = 0$$

Assuming that $\mathcal{D}$ outputs binary values for binary inputs, we get that:

$$\forall p \in \{0,1\}^* : \ c = \mathcal{D}(p') = \mathcal{D}(p) \in \{0,1\}^* \wedge c' = \mathtt{STEP}_{1/3}(c) = \mathcal{D}(p)$$

Since for such inputs the mask value is 0, we get that:

$$\forall p \in \{0,1\}^* : \ c^{masked} = c \Rightarrow \mathcal{D}_S(p) = \mathcal{D}(p)$$

Thus, for binary inputs, we get the correctness requirements that $\mathcal{D}_S$ outputs the same values as $\mathcal{D}$. We will now describe our formal security claim and its proof.

9.5 Security Proof

While the formal security proof may seem to be complicated, the idea behind it is very simple. Consider the n-dimensional cube of possible inputs. It can be

naturally divided into 2^n *orthants* with respect to the center of the cube, so that each orthant contains a unique binary point consisting of just zeroes and ones. What we show is that given any real valued input point p, the attacker can compute by himself the output of our secure implementation $\mathcal{D}_S(p)$ by just asking the original binary functionality $\mathcal{B}$ what is the value of $\mathcal{B}(p')$ for the unique binary point p' which resides in the same orthant as p (when p is at the boundary between orthants, the attacker outputs zeroes). Consequently, an oracle access to our secure implementation $\mathcal{D}_S$ does not leak any information about the secret key that is not already leaked by the original cryptogrpahic primitive. To prove this, we will formally show how to simulate the answer for any real-valued query for $\mathcal{D}_S$ when we are given blackbox access to the binary implementation that accepts only binary inputs.

Theorem 1 (Perfect Simulation). *Let $\mathcal{B}$ be the binary implementation of the cryptographic primitive such that for every binary input $\forall p \in \mathcal{P}_{\texttt{Bin}} : \mathcal{B}(p) = \mathcal{D}(p)$, and let $\mathcal{D}_S$ be our* secured *DNN-based implementation transformation of $\mathcal{D}$. For any probabilistic polynomial-time adversary $\mathcal{A}$ with oracle access to $\mathcal{D}_S$ (which we denote as $\mathcal{A}^{\mathcal{D}_S}$), there exist a probabilistic polynomial-time adversary $\mathcal{A}'$ with oracle access to $\mathcal{B}$ (which we denote as $\mathcal{A}'^{\mathcal{B}}$), that perfectly simulates $\mathcal{A}^{\mathcal{D}_S}$, such that the statistical distance between the output distribution of $\mathcal{A}$ and $\mathcal{A}'$ is zero.*

Formally,

$$\Delta(\mathcal{A}^{\mathcal{D}_S}, \mathcal{A}'^{\mathcal{B}}) = \frac{1}{2} \sum_{x} \left| \Pr[\mathcal{A}^{\mathcal{D}_S} = x] - \Pr[\mathcal{A}'^{\mathcal{B}} = x] \right| = 0.$$

where x can be any arbitrary function of the key.

Algorithm 8 Secure Blackbox Transformation Simulator

Input: A binary implementation of a keyed cryptographic primitive $\mathcal{B}$, a key $\texttt{k}$ and input p.
Output: output $c \in \mathcal{R}^N$.

1: **if** $p \notin P_{safe}$ **then** ▷ If the input is not "safe" return output of zeros.
2: **return** $\{0\}^N$
3: $p' \leftarrow \texttt{STEP}_{1/3}(p)$ ▷ Sanitization layer, applied on each input value
4: $\texttt{MASK}_{1/3} \leftarrow \Sigma_{p_i \in p}(\texttt{RECT}_{1/3}(p_i))$ ▷ Calculating masking value
5: $c \leftarrow \mathcal{B}(\texttt{k}, p')$
6: $c' \leftarrow \texttt{STEP}_{1/3}(c)$ ▷ Sanitization layer, applied on each output value
7: $c^{masked} \leftarrow \texttt{ReLU}(c' - \texttt{MASK}_{1/3})$ ▷ Masking layer, applied on each output value
8: **return** c^{masked}

Proof (Sketch). We will start by explaining our simulator that is described in Algorithm 8. The teal colored lines show the diff between the real secure implementation and our simulator. The only differences are added check that returns

an all zero output if $p \notin P_{safe}$ and the fact that we use the binary oracle $\mathcal{B}$ instead of the DNN-based implementation $\mathcal{D}$.

We will now prove that our simulator provides the same output as the secure implementation on all inputs. In our analysis we will handle two case. The first case is when the inputs are in the P_{safe} domain, and the second case is when the inputs are in the complementary "unsafe" $P_{unsafe} = P \setminus P_{safe}$ input domain. For each case, we will show that the output of our simulator is equal to the output of the secure implementation.

We start from the case where $p \notin P_{safe}$. From its definition:

$$\forall p \notin P_{safe} : \texttt{MASK}_{1/3} \geq 1$$

This means that in our secure implementation, due to the output sanitization and masking layers, for every such input, the output values will be all zero. Due to the check in line 1, our simulator will also return the same all zero for all such outputs.

For the complementary $p \in P_{safe}$, from its definition:

$$\forall p \in P_{safe} : p' \leftarrow \texttt{STEP}_{1/3}(p) \in \{0,1\}^*$$

I.e., for every such "safe" input, our approximate step function rounds the value to the nearest binary value. As we assume that for all such values $\mathcal{B}(p') = \mathcal{D}(p')$, and for such safe values the if statement at line 1 has no effect, our simulator will return the same value as our secure transformation.

Finally, as our simulator only has access to the binary oracle $\mathcal{B}$ and the input, we can use it to define $\mathcal{A}$'. $\mathcal{A}$' is just $\mathcal{A}$ where we replace the oracle queries to $\mathcal{D}_S$ with queries to our simulator which is based on calls to $\mathcal{B}$. Note that as our simulator only includes one call to $\mathcal{B}$, the query complexities for $\mathcal{A}$ and $\mathcal{A}$' is the same.

Security of Control Bits An interesting fine point in the formulation of the security properties is related to the fact that, in some cases, we may require another type of input value, which we call control bits. For example, to support both encryption and decryption, the encryption and decryption processes should be done either by providing two separate DNNs or by adding an additional control input bit z to the DNN, so that when $z = 0$ the DNN performs an encryption operation and when $z = 1$ it performs a decryption operation on the other input values. Whereas standard attacks on cryptosystems allow the attacker to either encrypt or decrypt chosen messages, the existence of such a z input enables the attacker to set it to an intermediate value such as $z = 0.5$, which will force the DNN to perform operations which are neither encryption nor decryption and whose meaning depends on the details of the DNN implementation. However, our protective techniques are sufficiently general to deal with such generalized attacks.

Security of Signature Verification A slightly different scenario we can consider is a DNN-based implementation of a digital signature verification scheme,

like the one considered in [9]. The main difference is that in this case the implementation only contains a public key; since there is no secret key to protect, our security guarantees are meaningless [6].

A standard signature verification scheme is an algorithm $\mathcal{A}$ which accepts as input a binary message m and a binary signature s: it should output 1 whenever the signature is valid, and 0 when it is not. Since we assume that the DNN implementation $\mathcal{A}'$ of $\mathcal{A}$ is correct, it should output the same 0/1 values for binary inputs. The security guarantee in this case should be that an adversary who cannot find a binary signature s which makes $\mathcal{A}(m, s) = 1$ should not be able to find any real-valued s' for which $\mathcal{A}'(m, s') = 1$. Notice that this also implies that the adversary should not be able to find one real-valued s'' for which $\mathcal{A}'(m, s'') > 1$ and another real-valued s''' for which $\mathcal{A}'(m, s''') < 1$, since by interpolation between s'' and s''' the adversary can also find an s' for which $\mathcal{A}'(m, s') = 1$ due to the continuity of $\mathcal{A}'$.

It is easy to verify that the secure transformation outlined in Figure 4 can also be used to protect against such attacks. In particular, we can guarantee that:

1. For all "unsafe" input values, the output will be 0, i.e., the verification fails.
2. For all binary inputs (and indeed for all inputs in P_0), we get the correct 0/1 results.
3. For all input values in $P_{safe} \setminus P_0$, our sanitization technique maps the inputs to a value which is strictly smaller than 1 and thus the verification fails.

10 Conclusion

In this paper we considered the basic problem of how to securely implement digital cryptography on an analog computer such as a ReLU-based DNN. After showing that all the natural implementations of cryptosystems as DNN's can be easily broken, we developed a new implementation technique which is provably secure and completely practical.

Acknowledgments. We would like to thank Orr Dunkelman, Nathan Keller, Eylon Yogev, Or Zamir, Francisco Rodríguez-Henríquez, and Nicholas Carlini for very fruitful discussions.
Eyal Ronen was supported in part by an ISF grant no. 1807/23, the Len Blavatnik and the Blavatnik Family Foundation, and the Stellar Development Foundation.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

[6] This issue was pointed out to us by Or Zamir.

A Attacking a More Generic Corner Definition

The corner function associated with corner $b = (b_1, \dots b_k)$, as defined in Definition 5, outputs value 1 at the vertex, outputs 0 for all points $(x_1, \dots x_k)$ that are at distance 1 or more from b, *i.e.,* such that $\sum_{i=1}^{k} |x_i - b_i| \geq 1$, and varies linearly in between. It is natural to wonder what happens if the separating hyperplane is made to be closer to the corner, and whether it would be sufficient to prevent the attacks we describe. For instance, we can define, for a distance $0 < c \leq 1$ and a corner b, the more general corner function:

$$\texttt{corner}_{c,b_1 \dots b_k}(x) \;=\; \texttt{ReLU}\left(\frac{1}{c}\left(\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} (1 - x_i) - k + c\right)\right),$$

which takes value 1 at the b, 0 for all points such that $\sum_{i=1}^{k} |x_i - b_i| \geq c$, and varies linearly in between. When such a corner function is used to build a natural implementation of the XOR function, it results in two *separated* `ReLUs`, as illustrated in Figure 7.

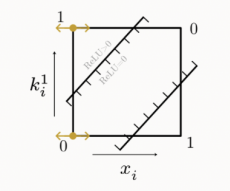


Fig. 7: A natural implementation of XOR based on separated `ReLUs`.

Our attacks for such implementations are very similar to the one described in section 7; in fact, the same symmetry-based attack applies directly. However, the existence of a buffer between the two ReLU's (where all inputs are sent to zero) enables a simpler variant of the attack.

In the case of separated ReLU's, we observe that tiny changes in the plaintext value x_i will be blocked at the two inactive corners of the square, and passed through at the two active corners of the square. This can reveal whether we are at the top side of the square or at the bottom side of the square, and thus expose the value of the i-th key bit $\texttt{k}_i$. For each value of bit $\texttt{k}_i$, the pair generation algorithm prepares two pairs of plaintexts that only vary in position i, where they take values $x_i^0 = 0$ and $x_i'^0 = \epsilon$, and $x_i^1 = 1$ and $x_i'^1 = 1 - \epsilon$, for some tiny ϵ.

Note that in this case we only use plaintext values which are already in the range $[0, 1]$, and thus forcing all the inputs to be in this range by using *ClippedReLU* does not stop the attack.

The corresponding pair generation algorithm, $\texttt{GenPairs}^{change}$, is described in Algorithm 9.

Algorithm 9 $\texttt{GenPairs}^{change}(p, i, \epsilon)$

$(x^0, x'^0) \leftarrow ((p|p_i = 0), (p|p_i = \epsilon))$
$(x^1, x'^1) \leftarrow ((p|p_i = 1), (p|p_i = 1 - \epsilon))$
return $(x^0, x'^0), (x^1, x'^1)$

We ran the corresponding attack on a natural implementation of the AES using separated ReLUs, with $c = 0.5$. The attack was run for 1000 random keys, which were all successfully recovered, 792 (79.2%) of which on the first try. In the remaining cases, a Fail was detected in one or several of the recovered key bits, so that 203 (20.3%) keys required a second base plaintext, and five (0.5%) required a third plaintext. These correspond to a total of 366 bit recovery failures, out of 128000 key bits recovered; it is important to note that all these initial failures were identified during the attack itself, rather than via comparison to the ground truth, and could thus be remedied by trying additional base plaintexts.

Analyzing these failures showed that they were all due to small changes around the preimage of zero through the Sbox being blocked in the first layer; increasing the magnitude of the change ϵ to 0.4 is sufficient to thwart such blocking, and results in the attack reaching 100% success rate.

B Neural AES Implementation in PyTorch

All variants of our neural AES are written in PyTorch and use only standard building blocks: affine layers and ReLU activations wired into a static computational graph. Unlike conventional networks, however, the weights are *not trained*; they are fixed, real-valued tensors obtained directly from the algebraic description of AES. As a result no back-propagation, optimiser state, or gradient tracking is needed.

For reproducibility, the complete source code is provided at

`https://github.com/DavidGerault/deep_neural_cryptography`

The `nn_aes.py` module in our GitHub repository implements one AES *round* as a fixed-weight feed-forward network in which every Boolean sub-operation is realised by a *corner-network* in the sense of Defs. 5–6.

The basic implementation, which we refer to as NeuralAESBase, performs the following operations for encryption:

- **Initial key addition.** The 128 input neurons are XORed to the 128 precomputed initial round key neurons. The XOR uses two corner neurons per input, and the output is 128 neurons.
- **Each of the 9 full rounds.**

  - **S-Box.** The 128 input neurons are expanded to $16 \cdot 256 = 4096$ intermediate corner neurons, and linearly mapped to 128 output neurons;
  - **Shiftrows-MixColumns.** The shiftrows rotation corresponds to a simple relabeling of the input neurons. The 128 inputs are expanded to $16 \cdot 2 \cdot 256 = 8192$ intermediate corner neurons for MUL2 and MUL3, followed by $128 \cdot 6 = 768$ intermediate neurons for the 4-way XORs.
  - **Round key addition.** The 128 input neurons are XORed to the 128 precomputed round key neurons. The XOR uses two corner neurons per input, and the output is 128 neurons.
- **S-Box.** The 128 input neurons are expanded to $16 \cdot 256 = 4096$ intermediate corner neurons, and linearly mapped to 128 output neurons;
- **Shiftrows-Round key addition.** The 128 input neurons are XORed to the 128 precomputed round key neurons. The XOR uses two corner neurons per input, and the output is 128 neurons.

In total, this corresponds to $256 + 128 + 9 \cdot (4096 + 128 + 8198 + 768 + 168 + 256 + 128) + 4096 + 128 + 256 + 128 = 128670$ neurons, arranged in $2 + 9 \cdot (2 + 3 + 2) + 2 + 2 = 69$ layers when the sums of corners are not merged with the next layer's linear part.

The decryption is similar in terms of parameters, but uses 512 additional corner neurons in the inverse MixColumns operation, due to the additional finite field multiplications with coefficients different from 1.

A well-known optimization [7], is to merge the S-Box and Mixcolumns step, by using *T-Tables* to map a byte x to $\mathrm{SB}(x), \mathrm{MUL}_2(\mathrm{SB}(x)), \mathrm{MUL}_3(\mathrm{SB}(x))$. The corresponding natural implementation, which we call TTablesNeuralAES, shaves 9 layers off the total count (one per MixColumns operation) and is, as a result, faster. On the other hand, the T-Tables technique is only applicable in the encryption direction, so the performance gain does not port to decryption.

The *protected* variants keep the core implementation untouched, but surrounds it with `STEP` and `RECT` at the start, and a subtraction in the plaintext. The application of `STEP` and `RECT` to the input adds in total 512 neurons (4 per input) in a layer prepended to the natural implementation (and 128 neurons for the summation, which can be merged into the initial add round key). The subtraction of the sum of `RECT`, followed by a ReLU, is added as a final layer.

C Benchmarking of Neural AES in Deep Neural Network Inference Pipelines

In the following, we first compare the information throughput (in bits per second) of our neural AES implementations in PyTorch (section B) and then benchmark their overhead in contemporary DNN inference pipelines.

Table 2 shows that our *provably secure* neural-cryptographic network, TTableNeuralAES, decrypts a batch of 10 000 AES blocks in 5.5 ms (throughput ≈ 234 Mbps) on an NVIDIA A100 GPU, matching its unsecured baseline; formal security machinery adds only negligible overhead. We now ask: *What is the latency overhead*

Implementation	Throughput
Encryption	
NeuralAESBase†	(310.86±2.01) Mbps
TTablesNeuralAES†	(365.06±2.98) Mbps
Secure TTablesNeuralAES‡	(364.87±2.52) Mbps
Decryption	
NeuralAESBase†	(234.05±1.25) Mbps
Secure NeuralAESBase‡	(234.07±1.18) Mbps

† PyTorch implementations as described in section B benchmarked with a batch size of 10 000 samples and 128-bit plaintexts per sample. Benchmarks were run on an A100 GPU.

‡ At a batch size of 10 000 samples the overhead of the protection layers is within the measurement error of the benchmarking. If we instead characterize the overhead at the level of a single AES block the added overhead by the protection layers is +1.1% in inference time, i.e., still negligible.

Table 2: Benchmarked throughputs of our neural AES implementations.

when such neural AES decryption precedes a contemporary deep neural network, such as a large-language model (LLM), in an inference pipeline? Because neural-AES layers are intended to sit in front of (or inside) larger models, we benchmark against the most popular open-source model (Meta AI's LLama) and one of the OpenAI GPT-2 family models and compare their runtimes to the neural AES decryption cost.

Model Selection and Loading We consider the GPT-2 Large and Llama 3.1 8B transformer models as provided by the Hugging Face `transformers` library [22]. The models are loaded in `torch.float16` precision to reduce memory footprint and improve inference speed relative to `float32`, reflecting common deployment practice.

Measurement Methodology Benchmarks are run on an NVIDIA A100 GPU. Precise runtimes are gathered with CUDA events (`torch.cuda.Event`) to avoid CPU–GPU synchronization artifacts. Each model is compiled with `torch.compile(mode="reduce-overhead")` to enable graph-level optimizations and kernel fusion (i.e., combining multiple operations into a single GPU kernel to reduce memory transfers and launch overhead). We perform five warm-up runs followed by five measured runs and report mean wall-clock inference times.
We selected 2,048 tokens as one benchmark to represent realistic encrypted document processing scenarios. This token count corresponds to approximately 1,463 words (using an empirical ratio of 1.4 tokens per word for English prose) or 2–3 pages of single-spaced business documents at standard formatting density (500

LLM Details		512 AES Blocks	
Model	**Params**	**LLM Time**	**vs AES**
GPT-2 Large	774M	27 ms	41×
Llama 3.1 8B	7.5B	1.68 s	2,557×

LLM times are measured on an NVIDIA A100 with `torch.compile` ("reduce-overhead"), using fp16 precision. The 512-block setup matches the byte volume of 2,048 LLM tokens; the "vs AES" columns compare against neural AES decryption times of 0.657 ms.

Table 3: Runtime comparison of neural AES and LLM inference speeds at matched input sizes corresponding to 512 AES blocks.

words per page). This length captures typical confidential communications such as reports, contracts, and financial statements that would require encryption in practice.
The input size of 2,048 LLM tokens is roughly equivalent to 512 AES blocks, due to an approximate 4:1 ratio between tokens and AES blocks.[7]

Results We tested the throughputs of neural AES versus contemporary language models. The fixed overhead of decrypting the input data is very small in comparison with the end-to-end latency of the considered models: Adding an AES decryption step in front of GPT-2 increases end-to-end latency by less than 3% for GPT-2 Large, and becomes negligible at the scale of Llama 3.1 8B. In other words, cryptographic protection is effectively *free* at the scale of LLM workloads.

References

1. Fips pub 197, advanced encryption standard (aes) (2001), u.S.Department of Commerce/National Institute of Standards and Technology
2. Abadi, M., Andersen, D.G.: Learning to protect communications with adversarial neural cryptography. arXiv preprint arXiv:1610.06918 (2016)

[7] The approximate 4:1 ratio between tokens and AES blocks is due to the following: On average, each token in common LLM tokenizers (e.g., GPT-2's BPE or LLaMA's SentencePiece) encodes 4–4.5 characters of English text[19,10], with each character typically occupying 1 byte in UTF-8 or ASCII encoding. Since an AES block operates on 16 bytes, this implies that 4 tokens (≈16 characters, respectively 16 bytes) map naturally to one AES block. Thus, 2,048 tokens correspond approximately to 512 AES blocks.

3. Basu, M.: China is betting on 'optical' computer chips – will they power ai? Nature (01 2026). https://doi.org/10.1038/d41586-026-00274-9
4. Bellini, E., Hambitzer, A., Protopapa, M., Rossi, M.: Limitations of the use of neural networks in black box cryptanalysis. In: International Conference on Information Technology and Communications Security. pp. 100–124. Springer (2021)
5. Coutinho, M., Albuquerque, R., Borges, F., García Villalba, L., Kim, T.H.: Learning perfectly secure cryptography to protect communications with adversarial neural cryptography. Sensors **18** (04 2018). https://doi.org/10.3390/s18051306
6. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems **2**(4), 303–314 (1989)
7. Daemen, J., Rijmen, V.: The design of Rijndael, vol. 2. Springer (2002)
8. Draguns, A., Gritsevskiy, A., Motwani, S.R., Rogers-Smith, C., Ladish, J., de Witt, C.S.: Unelicitable backdoors in language models via cryptographic transformer circuits. CoRR **abs/2406.02619** (2024). https://doi.org/10.48550/ARXIV.2406.02619, https://doi.org/10.48550/arXiv.2406.02619
9. Goldwasser, S., Kim, M.P., Vaikuntanathan, V., Zamir, O.: Planting undetectable backdoors in machine learning models. In: 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS). pp. 931–942. IEEE (2022)
10. Hoffmann, J.: Tokenization statistics and benchmarks for llms (2023), https://huggingface.co/docs/transformers/tokenizer_summary, retrieved from Hugging Face forums
11. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural networks **2**(5), 359–366 (1989)
12. Kalavasis, A., Karbasi, A., Oikonomou, A., Sotiraki, K., Velegkas, G., Zampetakis, M.: Injecting undetectable backdoors in obfuscated neural networks and language models. In: The Thirty-eighth Annual Conference on Neural Information Processing Systems (2024), https://openreview.net/forum?id=KyVBzkConO
13. Kanter, I., Kinzel, W., Kanter, E.: Secure exchange of information by synchronization of neural networks. EPL (Europhysics Letters) **57** (02 2002). https://doi.org/10.1209/epl/i2002-00552-9
14. Klimov, A., Mityagin, A., Shamir, A.: Analysis of neural cryptography. In: Advances in Cryptology—ASIACRYPT 2002: 8th International Conference on the Theory and Application of Cryptology and Information Security Queenstown, New Zealand, December 1–5, 2002 Proceedings 8. pp. 288–298. Springer (2002)
15. Mann, Z.Á., Weinert, C., Chabal, D., Bos, J.W.: Towards practical secure neural network inference: the journey so far and the road ahead. ACM Computing Surveys **56**(5), 1–37 (2023)
16. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics **5**, 115–133 (1943)
17. Minsky, M., Papert, S.: Perceptrons: An introduction to computational geometry. Cambridge tiass., HIT **479**(480), 104 (1969)
18. Pan, X., Srikumar, V.: Expressiveness of rectifier networks. In: Proceedings of The 33rd International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 48, pp. 2427–2435. PMLR (2016), https://proceedings.mlr.press/v48/panb16.html
19. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019), https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, openAI Technical Report

20. Rosenblatt, F.: The perceptron: A probabilistic model for information storage and organization in the brain [j]. Psychol. Review **65**, 386 – 408 (11 1958). https://doi.org/10.1037/h0042519
21. Wikipedia contributors: SIGSALY — Wikipedia, The Free Encyclopedia (2025), https://en.wikipedia.org/wiki/SIGSALY, [Online; accessed 12-February-2025]
22. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Transformers: State-of-the-art natural language processing. In: Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations. pp. 38–45 (2020)