Dreaming of Keys: Introducing the Phantom Gradient Attack

Åvald Åslaugson Sommervoll[a]
Institute of informatics, University of Oslo, Problemveien 7, 0315 Oslo, Norway

Keywords: Phantom Gradient Attack, Key Recovery, Neuro-cryptanalysis, Backpropagation, Neural Network, Replacement Function.

Abstract: We introduce a new cryptanalytical attack, the *phantom gradient attack*. The phantom gradient attack is a key recovery attack that draws its foundations from machine learning and backpropagation. This paper provides the first building block to a full phantom gradient attack by showing that it is effective on simple cryptographic functions. We also exemplify how the attack could be extended to attack some of ASCONs' permutations, the cryptosystem that won CAESAR the competition for authenticated encryption: security, applicability, and robustness.

1 INTRODUCTION

Neural networks have the past decade seen a wide array of academic and commercial applications. One notable exception is cryptography[1]. A reason is that neural networks rely on gradients of differentiable functions, while encryption and decryption typically rely on discrete functions. Our contribution is to replace these discrete functions with piecewise differentiable functions, thereby allowing for a neural network-based key-recovery. We dub this the phantom gradient attack, which aims to link the step-wise training of neural networks to key-recovery. The attack can be used to attack almost any cryptosystem. We attack some basic cryptographic functions and show how the attack could be extended to attack more complex cryptosystems like ASCON.

In 2015 Google released DeepDream, popularized the idea of "training"[2] the input using a pre-trained network. The *phantom gradient attack* builds on this idea by representing a cryptosystem as a neural network. This way, the cryptosystem acts as a pre-trained network, and we use it to train on our input. This training aims to recover the secret key. However, a lot of cryptographic functions are discrete and thereby do not have gradients. An essential part of our attack is to replace the discrete functions with piecewise differentiable ones. These functions have gradients, and we call these the *phantom gradients* of the original discrete function. The choice of the piecewise differentiable function is crucial, and we will refer to these functions as replacement functions[3]. Moreover, we will highlight some choices that correlate with successful attacks and state some general principles for good replacement functions.

In symmetric key encryption, there is a secret key, k, which is used for encryption and decryption. In this case, we can view the encryption as a function f_k and decryption as its inverse f_k^{-1}. Finding this f_k^{-1} is trivial if k is known, but it is intentionally hard if k is unknown. The phantom gradient attack presented in this paper attempts to recover this k. More specifically, the phantom gradient attack attempts to recover an input that would result in a specified output. In other words, given $f(x) = y$, it searches for an x^* such that $f(x^*) = y$, given that the function f and output y are known. If we look at our encryption we have:

$$Enc_k(p) = f_k(p) = f(k,p) = c, \tag{1}$$

where p is the plaintext and c is the ciphertext. However, as the plaintext is unknown in this case, we would recover both k^* and p^*. Furthermore, since $|k| + |p|$ is likely to be much larger than $|c|$, the recovered k^* would most likely[4] be different from k.

[a] https://orcid.org/0000-0001-5232-5630

[1] Neural networks have shown promise in side channel attacks, but not on an algorithmic level.

[2] We write "training" in quotation since we are updating on the input and not the weights.

[3] These replacement functions can often be viewed as extensions to their discrete counterpart, as they typically act the same for valid discrete inputs. However, this is not a requirement.

Sommervoll, Å.
Dreaming of Keys: Introducing the Phantom Gradient Attack.
DOI: 10.5220/0010317806190627
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 619-627
ISBN: 978-989-758-491-6

Therefore we assume that the plaintext is known so the plaintext can act as a constant. This way, we may only focus on finding a k^*. In order to find such a k^*, we have to take a closer look at the function[5] f_p. As already mentioned, we wish to represent this f_p as a neural network. To do this, we look at the individual functions that take part in the encryption and find piecewise differentiable functions to replace them. These replacement functions are of great importance, as their derivatives are what we use to recover the key.

The remaining paper is organized as follows: Section 2 discusses related work. Next, section 3 provides some details regarding implementations and an application of our attack on the XOR function. In section 4, we briefly introduce ASCON and its basic permutations, p_C, p_S, and p_L. We show that the input is easily recovered for the first two, whereas p_L is less susceptible to our phantom gradient attack. Finally, we conclude our findings in section 5 and cover possible future work in section 6.

2 RELATED WORK

Our *phantom gradient attack* has a clear connection to the field of neuro-cryptology. A field that was first formally described by Dourlens in his 1996 masters dissertation (Dourlens, 1996), where he described the possibility of neuro-cryptography and neuro-cryptanalysis. Since then, we have seen the addition of a neural cryptosystem in 2002 by Kinzel and Kanter (Kinzel and Kanter, 2002). They synchronized two neural networks by sending the networks' outputs through a public channel and training on them. Unfortunately, this cryptosystem was not completely secure, as Klimov et al. (Klimov et al., 2002) published a paper the same year that broke it three different ways. In neuro-cryptanalysis, Alini successfully applied an attack on DES and Triple-DES using neuro-Cryptanalysis in 2012s (Alani, 2012). He, like us, was working in the *known-plaintext* case. However, he is not interested in key-recovery. Instead, he simulates the decryption of DES and Triple-DES under a specific key. In this effort, his inputs are ciphertexts, and his reference outputs are plaintexts and train the weights accordingly. This procedure is in great contrast to our implementation, which trains no weights, uses the ciphertext as reference output, and a guessed key as input. His implementation required an average of 2^{11} plaintext ciphertext pairs for DES and 2^{12} for Triple-DES. In the phantom gradient attack implementation put forward in this paper, we only train on plaintext ciphertext pair, as we only want to recover a possible key. However, more training samples could help us avoid stagnation and ensure that the key recovered is the correct key; this may be fruit for future work. With his network trained to predict the ciphertext given the plaintext, he attempts to use his network to predict the ciphertext for new messages with some success[6]. Greydanus also attempts to use neural networks to simulate cryptosystems in his work, *Learning the Enigma with Recurrent Neural Networks*. This work exemplified some of the difficulty of simulating and learning a cipher with recurrent neural networks, even an outdated cryptosystem like Enigma (Greydanus, 2017). This work contributed to the *phantom gradient attack* introduced in this paper to only focus on a stateless FFNN representation instead of recurrent neural networks, which can be more memory efficient. Long before the popularization of Googles DeepDream in 1988 Lewis in his work *Creation By Refinement: A Creativity Paradigm for Gradient Descent Learning Networks* (Lewis, 1988) exemplified the idea of training on inputs. He trained a classification network to judge is a sequence of 5 music notes where valid or not. Then he used the trained network to generate music notes using backpropagation. Like Alani, (Alani, 2012), he first trains the network's weights, while the phantom gradients are predefined. This approach differs from the phantom gradient because his gradients are found through training of the neural network, while the phantom gradients are predefined. This definition gives the phantom gradients a larger degree of freedom, but at the cost of having perhaps unsuitable gradients. In terms of image generation and visualizations, there are many more works (Portilla and Simoncelli, 2000; Erhan et al., 2009; Simonyan et al., 2013). In these works, they always train on the entire input. However, our phantom gradient attack will often be used to attack only a specific part of the input. For example, in ASCON, we know a lot about the initial state of the sponge duplex construction (Dobraunig et al., 2016). Some techniques for generating adversarial examples also attack specific parts of the input, like *One Pixel Attack for Fooling Deep Neural Networks* by Su et al. There they change just one pixel in an image to fool a pre-trained network into misclassifying the image. *BriarPatches: Pixel-Space Interventions for Inducing Demographic Parity* by Gritsenko

[4]The phantom gradient attack could be fed multiple ciphertexts to increase this probability - more on this in section 6.

[5]The p is subscript because it is assumed to be constant and is not an argument for the function.

[6]The average number of wrong bits in the unseen pair is 8.3% for DES and 11.4% for Triple-DES.

et al. does something similar; however, their intervention is on a larger area of the image but constrained to be a small patch (Gritsenko et al., 2018). An alternative to representing the discrete cryptographic functions to continuous ones is to use the discrete functions, and train using binarized networks (Zhu et al., 2019). Networks that train on bit operations without proper gradients see considerable speedup compared to traditional networks, but at the cost of their accuracy.

3 IMPLEMENTATION AND RESULTS

Punishment. The loss function tells us if our training brings us closer to the actual output. However, it is not built into the loss function to take into account whether or not the predicted values are in the correct range. As we aim to recover bits, values larger than 1 and less than 0 are meaningless[7]. To prevent values from becoming increasingly negative or much greater than one, we introduce an additional punishment for such values. We choose a ridge regression like punishment measure: Our experiments found that a punishment closely related to that of a ridge regression worked well:

$$punish_{ridge}(x) = \begin{cases} \frac{1}{2}(x-1)^2 & \text{for } x > 1 \\ 0 & \text{for } 0 \leq x \leq 1 \\ \frac{1}{2}x^2 & \text{for } x < 0 \end{cases}. \quad (2)$$

This allows the learning to take values outside the range [0,1] but should help keep the values close to proper bit values. We also introduce a scalar λ_{punish}, which we use to adjust the punishment in relation to the loss.

Rounding. At the end of our run, the guessed key k^* typically consists exclusively of floating-point numbers. Therefore if we have reached our final iteration, we round the guessed key, k^*, to force it to assume integer values. This rounding at the end is primarily to polish the recovered key, but may in some cases, allow us to take the final leap to a candidate key k^*.

Momentum, Gradient Clipping and Decay. We may add momentum to our gradient descent by updating our input x_i like so: $x_i^{new} = x_i - \eta \cdot (\frac{\partial loss}{\partial x_i} + momentum \cdot \frac{\partial loss^{old}}{\partial x_i^{old}})$. Furthermore, to take incrementally smaller steps, we introduce a decay to the learning rate: Each iteration, the learning rate, *eta*, is updated: $\eta = \frac{\eta}{1+decay}$. This way, decay = 0 gives no decay. To avoid overly large gradients, we introduce a negative minimum gradient and a positive maximum gradient. We clip gradients smaller or larger than this threshold, a common technique to combat exploding gradients (Zhang et al., 2019).

Remap Input. Some initial experiments showed that even with ridge punishment, the inputs could be led astray by the phantom gradients. Furthermore, we found that typically with phantom gradients from eq. (10), if a bit became overly positive, its true value was typically 0. Similarly, if the bit value became overly negative, its true bit value was typically 1. To combat these stray gradients, we remap the inputs, so that overly positive bits are set to 0, and overly negative bits are set to 1. We define overly positive to be at 1.5 and overly negative to be at -0.5. This way, when we round at the end of the run, we force the network to make a valid guess restricted to valid bit values, and at the same time, we allow the bits to explore some values outside the valid range of 0 and 1. To indicate when this stricter boundary is used, we write that *remap* is true. We also tried input clipping, but this technique was much better at combating stagnation in the learning, as it also forces the algorithm to change its guess.

3.1 Phantom Gradient Attack on XOR

Practically all modern cryptosystems work exclusively on bits. Therefore, the use of binary functions in encryption is widespread. Perhaps most common is the XOR function, which takes two bits and returns their sum modulo 2. By itself, XOR can be used to provide *perfect security* (Shannon, 1949) by using the encryption function:

$$Enc_k(p) = k \oplus p = c, \quad (3)$$

where p is the plaintext, k is the key, and $\oplus$ is used to symbolize bitwise addition modulo 2. Each bit in the key k is random and independent of the other bits, with a 50 % probability of 0 and a 50% probability of 1. This provides *perfect security* since the probability of observing the ciphertext c is independent of the plaintext p, in other words: $P(c|p) = P(c)$. However, that does not mean that the plaintext p holds no significance. If we assume that the plaintext is known to the attacker, he can recover the key by computing $c \oplus p$. This trivial case where we know the plaintext p and the ciphertext c can also effectively be attacked

[7]Numbers between 0 and 1 can be interpreted as probabilities. Numbers above 0.5 may be viewed as it is more likely to be a one than a zero.

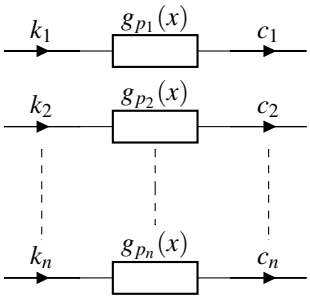
by the *phantom gradient attack*. This case can be represented as the network seen in fig. 1. Here g_{p_i} is used to represent our piecewise differentiable function that we use to represent XOR with the i-th bit value in the plaintext. A natural thought when choosing g_{p_i} is to let it be bitwise addition paired with a sine activation function to constrain it to modulo 2. However, the unpublished work by Parascandolo et al. (Parascandolo et al., 2016) showed some of the complications of learning with a sine activation function. Therefore we choose to instead separate XOR with constant 1 and XOR with the constant 0 into:

$$g_{p_i}(x) = \begin{cases} 1-x & \text{for } p_i = 1 \\ x & \text{for } p_i = 0. \end{cases} \tag{4}$$

It must be stressed that this choice is just one among many. For any gradient descent, we need a loss function; for this paper, we will use a square error:

$$loss = \frac{1}{2}\sum_{i}^{n}(c_i^* - c_i)^2, \tag{5}$$

where c_i^* is the predicted output bit in the i-th position, and c_i is the true bit value of the ciphertext in the i-th position. With this replacement function, this loss function, and input a, and learning rate $\eta = 1$, this replacement function can always recover the full key in one step. Formally, by recovering the full key, we mean that all the bits in the key are correct; similarly, if one or more bits are incorrect, the full key has not been recovered. Since all inputs are independent we can illustrate all possible outcomes by letting $\vec{p} = [1,0,1,0]$ and the targets be $\vec{c} = [1,1,0,0]$. Then we can construct the neural network based on fig. 1 and eq. (4). On such a network, a single iteration



where n is the number of bits in the plaintext p, and g_{p_i} is the reduction of f_p that only works on a single bit instead of a bit sequence.

Figure 1: XOR with a constant as a FFNN.

would be:

$$k_0^* = k_0^* - \eta \cdot \frac{\partial loss}{\partial k_0^*} = a - 1 \cdot \frac{\partial loss}{\partial c_0^*}\frac{\partial c_0^*}{\partial k_0^*} = 0 \tag{6}$$

$$k_1^* = k_1^* - \eta \cdot \frac{\partial loss}{\partial k_1^*} = a - 1 \cdot \frac{\partial loss}{\partial c_1^*}\frac{\partial c_1^*}{\partial k_1^*} = 1 \tag{7}$$

$$k_2^* = k_2^* - \eta \cdot \frac{\partial loss}{\partial k_2^*} = a - 1 \cdot \frac{\partial loss}{\partial c_2^*}\frac{\partial c_2^*}{\partial k_2^*} = 1 \tag{8}$$

$$k_4^* = k_4^* - \eta \cdot \frac{\partial loss}{\partial k_4^*} = a - 1 \cdot \frac{\partial loss}{\partial c_4^*}\frac{\partial c_4^*}{\partial k_4^*} = 0. \tag{9}$$

We observe that the recovered $\vec{k^*}$ is correct and was found independently from the initial input a. The key can also found with an η smaller than 1; this would just take more iterations.

3.2 XOR between Two Inputs

XOR between two inputs is also common in modern cryptosystems, especially in the construction of S-boxes[8]. Like previously, we have to represent XOR as a piecewise continuous function. One approach is to build on the previous replacement function and create the nonlinear function:

$$f(x,y) = x + y - 2xy, \tag{10}$$

which has all the desired XOR properties, and it collapses to the cases in eq. (4) if one of the bits in question are constant. The derivatives of this function is $\frac{\partial f}{\partial x} = 1 - 2y$ and $\frac{\partial f}{\partial y} = 1 - 2x$, which means that the gradient is 0 for $x = \frac{1}{2}$ or $y = \frac{1}{2}$. The vanishing gradients at 0.5 is a potential weakness as this value may act as a barrier preventing movement from values below 0.5 to move above 0.5 and vice versa. A way to address this concern is to have gradient descent with momentum. Additionally, the full gradient may not be 0 at 0.5 since the loss typically depends on many outputs, such as out1 and out2 in eq. (12).

Example: The simplest example, in this case, is just two bits as input, which are XOR-ed, as shown in fig. 2. As in section 3.1, we want to train on the initial

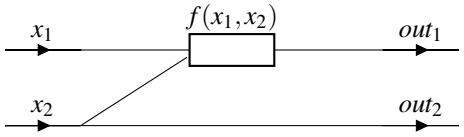
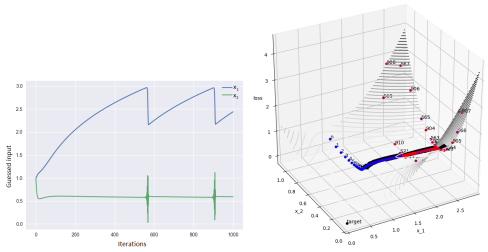
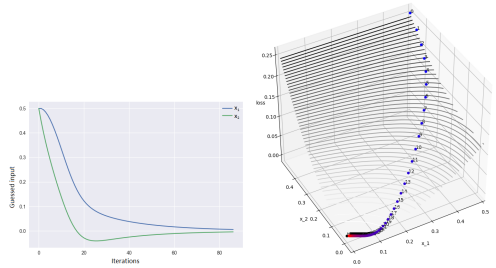


Figure 2: Example FFNN for XOR between two inputs.

[8]S-boxes stands for substitution boxes, and are often computed by a network so that the substitution can go fast in hardware.



(a) Input history $(1,1)$ (b) Contour plot $(1,1)$



(c) Input history $(\frac{1}{2},\frac{1}{2})$ (d) Contour plot $(\frac{1}{2},\frac{1}{2})$

Figure 3: XOR between inputs learning success. (a)+(c): The y-axis gives the guessed input bit, and the x-axis counts the number of iterations. The blue line is for the guessed bit for x_1, and the green line gives the guessed bit for x_2. (b)+(d): In the contour plot, we have plotted x_1, x_2, and the loss against each other. Each dot corresponds to a guess, and the iteration number of the guess is written next to the dot. As the number of iterations increases, the dots color change from blue to red, and the target is shown as a black dot.

guessed inputs; we call these inputs x_1, x_2. We see that x_1 is xor-ed with x_2, while x_2 is left unaltered, meaning that we get the following gradients:

$$\frac{\partial loss}{\partial x_1} = \frac{\partial loss}{\partial out1} \cdot \frac{\partial out1}{\partial x_1} \tag{11}$$

$$\frac{\partial loss}{\partial x_2} = \frac{\partial loss}{\partial out1} \cdot \frac{\partial out1}{\partial x_2} + \frac{\partial loss}{\partial out2} \cdot \frac{\partial out2}{\partial x_2}. \tag{12}$$

It must be noted that even in this simple example, of phantom gradient attack can fail. If try to recover $x_1 = x_2 = 0$, and start with initially random x_1 and x_2 we get a recovery rate of 96% (9599 out of 10000). In other words, the starting point can hold great significance for the success of our attack. To analyze this, we look at two cases, initial input [1,1] and [0.5, 0.5], as can be seen in fig. 3. We see that in fig. 3b, the phantom gradients lead the input astray, and it gets stuck in a repeating pattern. However, with a better starting point like [0.5,0.5], learning is easy, and the solution is found almost instantly. A possible pitfall may be that the phantom gradients lead our guesses astray by moving them outside the range of 0 and 1; in section 6, we discuss ways to prevent this.

4 ATTACK ON ASCON's UNDERLYING FUNCTIONS

ASCON is a cryptography system for lightweight authenticated encryption and hashing. It has entered two competitions:

1. The *Competition for Authenticated Encryption: Security, Applicability, and Robustness* (*CAESAR*) (Dobraunig et al., 2016).
2. NIST's *Lightweight Cryptography* standardization competition (Dobraunig et al., 2019).

So far in the competitions, it has won CAESAR (Bernstein, 2019) and is currently a third-round qualifier of the NIST standardization competition (NIST, 2020). ASCON has many different versions; for this paper, we will investigate its most current iteration, ASCON v1.2. Furthermore, within ASCON v1.2, there are some variants. We will only be looking at encryption and decryption using ASCON-128 within ASCON v1.2. From this point on, when we refer to ASCON encryption and ASCON permutation, we refer to them as they are in ASCON-128 v1.2, details in table 1. Full ASCON encryption uses a secret state of 320 bits that undergo a series of permutations. Only 64 bits are observed before the state is permuted again. This segmentation of the observed output means that if one were to attack the ASCON encryption using the phantom gradient attack, we would only get gradients from 64 bits to attack a 128-bit key. We can use additional 64-bit blocks, recover possible intermediate states, and work backward from these possible intermediate states. However, in this paper, we will only be looking at ASCON's three permutations; p_C, p_S, and p_L. To clarify the individual steps, we divide p_S into p_{S_1}, p_{S_2}, and p_{S_3}. Furthermore, when running our neural networks, we use the settings seen in table 2.

4.1 p_C Permutation

The first permutation in ASCON is the p_C permutation, which only consists of an XOR with a constant[9].

Table 1: ASCON-128 specifications.

Number of bits					# rounds	
key	nonce	tag	S_r	S_c	p^a	p^b
128	128	128	64	256	12	6

(This table is heavily influenced by table 1 in the ASCON v1.2 submission to CAESAR (Dobraunig et al., 2016).)

[9]This constant varies with p^b and p^a and how many permutations that have taken place.

In section 3.1, we saw that this could be easily solved using the phantom gradient attack.

4.2 p_S Permutation

The p_S permutation defines a 5-bit substitution. As it only works on 5 independent bits, we can reduce the problem from 320 bits down to 5 without losing any complexity. This reduction allows us to check phantom gradient attacks recovery capabilities on any of the possible 32 (2^5) different inputs. This substitution can be expressed as a series of XOR-, AND- and NOT- gates. To further simplify this network, we divide it into three parts p_{S_1}, p_{S_2}, and p_{S_3}, as shown in fig. 4.

4.2.1 p_{S_1} Permutation

For p_{S_1} we have the following mapping:

$$p_{S_1}\left(\begin{bmatrix}x_0\\x_1\\x_2\\x_3\\x_4\end{bmatrix}\right)=\begin{bmatrix}x_0\oplus x_4\\x_1\\x_2\oplus x_1\\x_3\\x_4\oplus x_3\end{bmatrix}.$$

The p_{S_1} permutation only uses three XOR's, all[10] of which we can represent with eq. (10). With phantom gradients from eq. (10) and settings as in table 2, we recover the input in all 32 cases.

Table 2: Settings for backpropagation.

parameter	p_C	p_{S_1}	p_{S_2} and p_{S_3}	p_S	Σ_1 and Σ_2
η	1	0.01	0.2	0.02	0.2
momentum	0	0.01	$2e$-3	0.2	0.9
decay	0	10^{-3}	10^{-4}	10^{-9}	1
max gradient	∞	∞	7	7	7
min gradient	$-\infty$	$-\infty$	-7	-7	-7
λ_{punish}	0	0	$4e$-3	0.04	0.04
remap	False	False	False	True	True
Initial input	$\{\frac{1}{2}\}^5$	$\{\frac{1}{2}\}^5$	$\{\frac{1}{2}\}^5$	.4, .6	na
Iterations	1000	1000	1000	1000	1000

4.2.2 p_{S_2} Permutation

The p_{S_2} permutation can be expressed as:

$$p_{S_2}\left(\begin{bmatrix}x_0\\x_1\\x_2\\x_3\\x_4\end{bmatrix}\right)=\begin{bmatrix}x_0\oplus(NOT(x_1)\cdot x_2)\\x_1\oplus(NOT(x_2)\cdot x_3)\\x_2\oplus(NOT(x_3)\cdot x_4)\\x_3\oplus(NOT(x_4)\cdot x_0)\\x_4\oplus(NOT(x_0)\cdot x_1)\end{bmatrix}.$$

We replace the NOT gate[11] with $1-x_1$, and the $\oplus$ function with eq. (10):

$$f(x_i,x_j,x_k)=x_i+(1-x_j)*x_k-2*x_i*(1-x_j)*x_k, \quad (13)$$

where $j=i+1(mod5)$ and $k=i+2(mod5)$. This means that bitwise rotations should act equivalently, that is a bit sequence $[b_0,b_1,b_2,b_3,b_4]$ should behave similarly to $[b_1,b_2,b_3,b_4,b_0]$, $[b_2,b_3,b_4,b_0,b_1]$, $[b_3,b_4,b_0,b_1,b_2]$ and $[b_4,b_0,b_1,b_2,b_3]$. The equivalent permutation groups are shown in table 3. To achieve full key recovery for any key, we use the settings as seen in table 2. All the inputs that belong to the same group recovered their bit sequence after the same number of iterations. However, perhaps surprisingly, group 4 and group 6 need 199 and 159 iterations, while the slowest of the remaining groups finish in 48 iterations. This wide gap is a little surprising. It can be related to the fact that groups 4 and 6 are the two groups containing the only two-bit alternating sequences: 01010 and 10101. This fact may be a coincidence, but it seems like our phantom gradients struggle a little with such alternating bit sequences at p_{S_2}.

4.2.3 p_{S_3} Permutation

The p_{S_3} is defined as:

$$p_{S_3}\left(\begin{bmatrix}x_0\\x_1\\x_2\\x_3\\x_4\end{bmatrix}\right)=\begin{bmatrix}x_0\oplus x_4\\x_1\oplus x_0\\1\ -x_2\\x_3\oplus x_2\\x_4\end{bmatrix}$$

We see that this permutation only consists of previously defined functions: XOR between two indices,

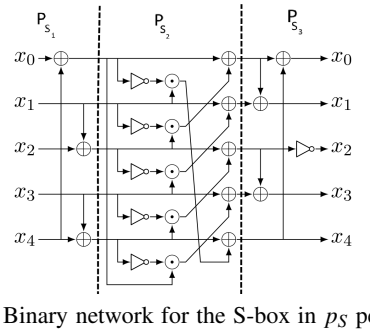


Figure 4: Binary network for the S-box in p_S permutation divided into p_{S_1}, p_{S_2}, and p_{S_3}.

[10] We just have to make sure that $x_4\oplus x_3$ happens after $x_0\oplus x_4$.

[11] Note that this is the same as our replacement function of XOR with 1 in eq. (4).

eq. (10), and NOT (XOR with 1, eq. (4)). We achieve full key recovery[12] by reusing the settings p_{S_2}, table 2. The maximum number of iterations required for our attack on p_{S_3} is higher than the worst-case we observed for group 4 in p_{S_2}. This is as expected as p_{S_3} is a simpler permutation. However, perhaps surprising is that the smallest number of iterations required for p_{S_3}, 58, is higher than the smallest number of iterations required for p_{S_2}, 12.

The full p_S permutation is, of course, more complicated than its components. However, we achieve full key recovery using the settings seen in table 2. The most notable difference is that we no longer guess $[\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}]$ as the gradient is zero for this input. We, therefore, assume that like in ASCON that x_0, x_3, and x_4 are known,[13] and we only recover x_1 and x_2.

4.3 p_L Permutation

The p_L permutation is a combination of bitwise rotation and a three-way XOR on each 64-bit block. In this paper, we will only be looking at Σ_1 and Σ_2 as they affect the same blocks as the key started in. However, all the blocks are treated similarly. Based on eq. (10) we create the following formula for this three-way XOR:

$$f(x,y,z) = x + y + z - 2xy - 2xz - 2yz + 4xyz \quad (14)$$

which means that for Σ_1 and Σ_2 we get:

$$\Sigma_1 : f\left(x_{1,i}, x_{1,(i+61(mod64))}, x_{1,(i+39(mod64))}\right)$$
$$\Sigma_2 : f\left(x_{2,i}, x_{2,(i+01(mod64))}, x_{2,(i+06(mod64))}\right)$$

In contrast to earlier XOR examples, all the bits are affected by XOR at the same time. This means that the weakness of the vanishing derivative at 0.5 is even more of an obstacle. Therefore we do two things to aid the learning: 1. We let the initial η be large to build momentum initially, but we have a large decay so that it only moves fast in the beginning. To ensure that we have some learning rate for later iterations, we bound the minimal η value to a small value. In this case, we set the boundary to $\eta_{min} = 0.02$. 2: To help cross during later iterations, we choose a random index that is closer than some ε_{xor} to 0.5. Then we add:

$$\lambda_{xor} \cdot sign\left(\frac{\partial f(x_i, x_{i\boxplus 61}, x_{i\boxplus 39})}{\partial x_i}\right), \quad (15)$$

to the diagonal position corresponding to this index, where λ_{xor} is a predefined constant and $\boxplus$ symbolizes addition under modulo 64. The other matrix cells that impact this input are scaled-down by with λ_{xor} to ensure that 0.5 avoided. We call this a *gradient jump*, and we set ε_{xor} to 0.01 and λ_{xor} to 5. In contrast to the p_C permutation, where we proved that we could always recover the input, and the p_S permutation where we could test for all 32 possible inputs, we cannot test for all $2^{64} \approx 10^{19}$ possible inputs. Furthermore, we do not achieve full key recovery on Σ_1 and Σ_2. To analyze our performance on these permutations, we reduce the complexity by dropping leading bits. This way, we can adjust the number of bits to be between 1 bit and 64 bits. To analyze our performance, we start doing a 100 runs on 1 bit and iteratively increase the number of bits until we reach the full 64 bits. We use the settings as seen in table 2, where our initial guess has the same number of bits we wish to recover. Each element in our initial guess is randomly chosen to be either 0.4 or 0.6, as our initial experiments showed that this improved performance. For both Σ_1 and Σ_2, we do this with and without *gradient jump*.For almost all runs, the algorithm performs better with *gradient jump*. However, both of them perform poorly and have 0 successes on the full 64 bits. So even the *gradient jump* could not properly compensate for this suboptimal gradient. There is room for future work to investigate replacement functions that provide better phantom gradients.

Table 3: p_{S_2} permutation groups.

group1	group2	group3	group4
00000	00001	00011	00101
	10000	10001	10010
	01000	11000	01001
	00100	01100	10100
	00010	00110	01010
group5	group6	group7	group8
00111	01011	01111	11111
10011	10101	10111	
11001	11010	11011	
11100	01101	11101	
01110	10110	11110	

[12]Full key recovery means that all the bits in guessed key are correct.

[13]x_0 is a constant and $[x_3, x_4]$ are nonces, like a timestamp.

5 CONCLUSION

We have shown that the phantom gradient attack works on simple cryptographic functions. It also shows some promise on attacking ASCON's permutations, but as used in this paper, the attack is unsuccessful on ASCON's third permutation p_L. The two other permutations, p_C and p_S, were effectively attacked. The phantom gradient attacks failure on p_L is likely our replacement functions whose gradients are 0 at $\frac{1}{2}$

for XOR. It must be stressed that there is nothing inherently different from p_L, which renders it immune to the phantom gradient attack. It is most probably a question of finding the correct work around this "one-half"-challenge. These first results hold promise, as it shows that gradual learning of neural networks can also be applied to key recovery in cryptology.

6 FUTURE WORK

There is much room for future work on the phantom gradient attack. In particular, research regarding good replacement functions. Ideally, the replacement function should keep as many as the properties of traditional XOR. For example: $(x \oplus y) \oplus x$ should ideally be y in the replacement function as well. More generally, there is much room for attempting to attack other cryptosystems. For example, if we use the phantom gradient attack to attack a public cryptography scheme, we can use the public key to generate as many training samples as needed. Then we can use the phantom gradient attack to attack the decryption function: $f_c(k_{private}) = p$, The subscript c is the generated ciphertext, $k_{private}$ is the secret private key, and p is the chosen plaintext. We subscript c since, for each iteration, we assume that it is constant like we did with the plaintext in this work. The attack may also be extended even to work when the plaintext is unknown; however, this will likely require many training samples. As the phantom gradient attack is a new cryptanalytical attack, there is room for studying how to protect against it. Since it draws its foundation from neural networks, one could draw from cases where neural networks struggle. For example, learning works better on deep networks rather than wide networks. A cryptosystem that has to be represented as a wide network may be less vulnerable to a phantom gradient attack. For training the network, we tried *gradient descent* and *gradient descent with momentum* in this paper. However, other optimizers remain untested. Two natural candidates are the neural network optimizers ADAM and RMSProp. Moreover, it is not obvious that square error is the most suited loss function. Testing different optimizers and loss functions are low hanging fruits for future research.

ACKNOWLEDGMENTS

The author wishes to give a special thanks to Audun Jøsang and Thomas Gregersen for valuable discussion and words of encouragement.

REFERENCES

Alani, M. M. (2012). Neuro-cryptanalysis of des and triple-des. In *International Conference on Neural Information Processing*, pages 637–646. Springer.

Bernstein, D. J. (2019). Crypto competitions: Caesar submissions. https://competitions.cr.yp.to/caesar-submissions.html. (Accessed on 03/19/2020).

Dobraunig, C., Eichlseder, M., Mendel, F., and Schläffer, M. (2016). Ascon v1.2. Submission to Round 3 of the CAESAR competition.

Dobraunig, C., Eichlseder, M., Mendel, F., and Schläffer, M. (2019). Ascon v1.2. Submission to Round 1 of the NIST Lightweight Cryptography project.

Dourlens, S. (1996). Applied neuro-cryptography and neuro-cryptanalysis of des. Master Thesis. Advisor: Riesner, Christian.

Erhan, D., Bengio, Y., Courville, A., and Vincent, P. (2009). Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1.

Greydanus, S. (2017). Learning the enigma with recurrent neural networks. *arXiv preprint arXiv:1708.07576*.

Gritsenko, A. A., D'Amour, A., Atwood, J., Halpern, Y., and Sculley, D. (2018). Briarpatches: Pixel-space interventions for inducing demographic parity. *arXiv preprint arXiv:1812.06869*.

Kinzel, W. and Kanter, I. (2002). Neural cryptography. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.*, volume 3, pages 1351–1354. IEEE.

Klimov, A., Mityagin, A., and Shamir, A. (2002). Analysis of neural cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 288–298. Springer.

Lewis, J. P. (1988). Creation by refinement: a creativity paradigm for gradient descent learning networks. In *ICNN*, pages 229–233.

NIST (2020). Lightweight cryptography | csrc. https://csrc.nist.gov/projects/lightweight-cryptography. (Accessed on 03/19/2020).

Parascandolo, G., Huttunen, H., and Virtanen, T. (2016). Taming the waves: sine as activation function in deep neural networks.

Portilla, J. and Simoncelli, E. P. (2000). A parametric texture model based on joint statistics of complex wavelet coefficients. *International journal of computer vision*, 40(1):49–70.

Shannon, C. E. (1949). Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*.

Zhang, J., He, T., Sra, S., and Jadbabaie, A. (2019). Analysis of gradient clipping and adaptive scaling with a relaxed smoothness condition. *arXiv preprint arXiv:1905.11881*.

Zhu, S., Dong, X., and Su, H. (2019). Binary ensemble neural network: More bits per network or more networks per bit? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.