Evaluating the Security of Logic Locking on Deep Neural Networks

You Li*
you.li@u.northwestern.edu
Northwestern University
Evanston, IL, USA

Guannan Zhao*
gnzhao@u.northwestern.edu
Northwestern University
Evanston, IL, USA

Yunqi He
yunqi.he@u.northwestern.edu
Northwestern University
Evanston, IL, USA

Hai Zhou
haizhou@northwestern.edu
Northwestern University
Evanston, IL, USA

ABSTRACT

Deep neural networks are susceptible to model piracy and adversarial attacks when malicious end-users have full access to the model parameters. Recently, a logic locking scheme called HPNN has been proposed. HPNN utilizes hardware root-of-trust to prevent end-users from accessing the model parameters. This paper investigates whether logic locking is secure on deep neural networks. Specifically, it presents a systematic I/O attack that combines algebraic and learning-based approaches. This attack incrementally extracts key values from the network to minimize sample complexity. Besides, it employs a rigorous procedure to ensure the correctness of the extracted key values. Our experiments demonstrate the accuracy and efficiency of this attack on large networks with complex architectures. Consequently, we conclude that HPNN-style logic locking and its variants we can foresee are insecure on deep neural networks.

CCS CONCEPTS

• **Security and privacy → Security in hardware**.

KEYWORDS

IP protection, Logic locking, Reverse engineering neural networks

1 INTRODUCTION

Deep neural networks (DNNs) have achieved remarkable success in various applications. However, DNN models are suffering from intellectual property (IP) piracy and security threats. Adversaries are motivated to extract the parameters of a DNN model for two main reasons. Firstly, training a new model is expensive. *a)* Deep learning is data-hungry, while the data can be proprietary or require enormous efforts to collect; *b)* the expertise in tweaking the architecture, the learning algorithm, and the hyper-parameters is essential for deep learning; *c)* the training process demands substantial computational resources and a long time. Secondly, the exposure of the model parameters poses severe threats to the security and confidentiality of that model. With access to a leaked model, an adversary can launch *a)* the evasion attack [5] and the data poisoning attack [26] to abuse or deceive the victim model, or *b)* the inversion attack [9] to reconstruct sensitive training examples from the model.

Nowadays, many AI tasks are executed on dedicated hardware accelerators, such as TPU, NPU, GPGPU, and acceleration modules. Accordingly, *hardware root-of-trust* has been adopted to protect intellectual properties and mitigate vulnerabilities in DNNs. In this setting, a secret *key* is stored on hardware in a tamper-proof memory [3], a Trusted Platform Module (TPM) [20], or a Hardware Security Module (HSM) [25]. An adversary cannot retrieve the stored key directly, even with physical access to the hardware. An incorrect key will significantly alter the functionality of the DNN model, thus reducing the prediction accuracy and thwarting information leakage. A combination of a hardware accelerator, a key storage module, and a pre-stored key serves as a copy of license to access the service provided by the IP owner. A DNN model and its future parameter updates can be released publicly on cloud platforms. Only the subscribers who have acquired valid licenses can fully restore the functionality of the DNN model.

Logic locking [24, 27] is a prominent and well-established technique to safeguard hardware designs. It embeds binary key bits into an integrated circuit's netlist (a network of logic gates). The original functionality of that circuit can be restored only when a correct key is inserted. Logic locking achieves four objectives simultaneously [32]: *i) locking robustness* ensures that inferring a correct key is prohibitively expensive; *ii) structural robustness* means that removing or bypassing the locking module is impossible; *iii) locking efficiency* implies that the locking scheme introduces only a minimal overhead and requires only a small number of key bits; *iv) end-to-end protection* guarantees that the design is protected from all parties in the supply chain.

Hardware Protected Neural Network (HPNN) [7] is a pioneering work that applies the methodology of logic locking on DNN models. HPNN selects a small subset of neurons in the hidden layers of the DNN as the *key-protected neurons*. A key bit is associated with every such neuron, controlling whether to *flip* the sign of the pre-activation value. A DNN model is trained as a function of a pre-selected key pattern. In this way, model parameters and key bits are closely entangled. HPNN has addressed the limitations of several existing techniques. For example, DNN watermarking [1, 11] cannot prevent illegal private uses. Input, output, and convolutional kernel obfuscations [10, 18] are susceptible to removal attacks and unauthorized reproduction by end users, because their key bits are not entangled with learnable parameters. General public-key encryption on model parameters [30] will incur huge overheads.

It is demonstrated that reverse engineering HPNN through training is not worthwhile: starting from a random incorrect key, an adversary has to spend even more computational resources to fine-tune the model than training one from scratch [7]. Other approximate attacks, including a well-crafted attack based on genetic algorithm [2], achieve low accuracy on HPNN-encrypted networks. However, our study reveals that an adversary can directly recover the correct key with an I/O attack. Specifically, a malicious end-user can query a working model it possesses with carefully selected input examples, and thus infer the associated key bits from the outputs. In this way, it can replicate the model without the permission of the IP owner.

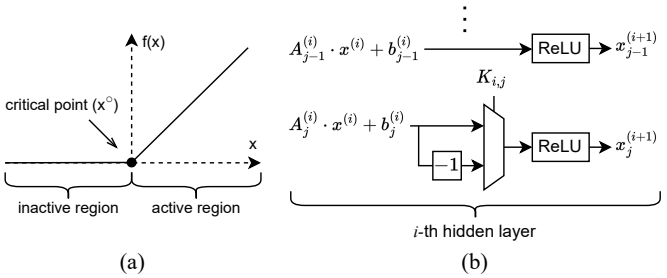
This paper presents a systematic attack algorithm against HPNN and other feasible encryption schemes we can foresee that use logic locking. It leverages the hierarchical nature of DNNs and the symmetric property of the ReLU activation function to reduce sample complexity significantly. In addition, it employs a rigorous validation and correction mechanism to ensure the correctness of the extracted key. Experiments show that our attack algorithm can scale to large DNNs and generalize to complex network architectures.

*Equal contribution.

This work is partially supported by the NSF under grants 2113704 and 2148177.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3658248



(a) (b)

Figure 1: (a) The Rectified Linear Unit (ReLU) activation function. The function is at the critical point if its input equals 0. (b) An unprotected neuron (upper) and a protected neuron (lower) of HPNN. The inserted flipping unit negates the pre-activation value of the protected neuron when the key value equals 1.

Our main contributions are:

• We establish a theoretical framework for security analysis of logic-encrypted deep ReLU networks;

• We develop an algebraic approach to infer the value of a single key bit when the network is contractive;

• We propose a learning-based approach as well as a validation and correction procedure to extract the key value of a hidden layer when the network is expansive;

• We evaluate the accuracy, scalability, and generality of the attack algorithm on practical networks.

2 BACKGROUND

2.1 Preliminaries

A DNN model can be represented as a function $f : \mathcal{X} \to \mathcal{Y}$, which takes inputs from the input space $\mathcal{X} \subseteq \mathbb{R}^P$ and returns outputs to the output space $\mathcal{Y} \subseteq \mathbb{R}^Q$. A k-layer deep neural network [6] f is an alternating sequence of linear transformations and a total of k non-linear activation functions: $f = f_{k+1} \circ \sigma \circ f_k \circ \cdots \circ \sigma \circ f_1$. In this equation, the i-th *hidden layer* is given by a linear transformation f_i followed by an element-wise ReLU activation function σ. Specifically, $f_i(x_{i-1}) = A^{(i)}x_{i-1}+b^{(i)}$ is an affine transformation, in which the *post-activation hidden state* $x_{i-1} \in \mathbb{R}^{d_{i-1}}$ is a d_{i-1}-dimensional vector, the *weights* $A^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$ is a d_i by d_{i-1} matrix, and the *biases* $b_i \in \mathbb{R}^{d_i}$ is a d_i-dimensional vector. Notice that the final *output layer* f_{k+1} is not followed by activation functions. All parameters, represented as $A^{(i)}$ and $b^{(i)}$ for $i \in 1, \cdots, k+1$, can be updated during training.

The above formulation assumes that all the hidden layers are fully connected layers. In reality, a neural network may also contain convolutional and pooling layers, residual connections [12], and attention units [28], but such a network can also be represented as linear transformations locally. Additionally, a *softmax* layer can be attached to the output layer. In this case, given an input example $x \in \mathcal{X}$, we call x_{k+1} the *logits* of $f(x)$ and $y = softmax(x_{k+1})$ the output vector of $f(x)$.

Each component of σ is a ReLU activation function [17] defined as $\phi(z) = \max(z, 0)$. ReLU has established itself as the default choice for deep learning because DNNs with ReLUs can be optimized more efficiently [21]. As shown in Figure 1(a), a ReLU is a piecewise linear function. We denote the j-th *neuron* in the i-th hidden layer as $\eta_{i,j}$. It computes $x_{i,j} = \phi(A_j^{(i)}x_{i-1} + b_j^{(i)})$, where $A_j^{(i)}$ is the j-th row of $A^{(i)}$ and $b_j^{(i)}$ is the j-th element of $b^{(i)}$. Particularly, we refer to $z_{i,j} = A_j^{(i)}x_{i-1} + b_j^{(i)}$ as the *pre-activation value* of $\eta_{i,j}$. We say that $\eta_{i,j}$ is at its *critical point* if $z_{i,j} = 0$. Moreover, a neuron is *inactive* if $z_{i,j} \leq 0$ and is *active* if $z_{i,j} > 0$.

2.2 HPNN Implementation

The actual implementation of HPNN is depicted in Figure 1(b). HPNN embeds several flipping units into a DNN model or the underlying hardware accelerator. Every unit modifies the pre-activation value of the associated neuron:

$$z_{i,j} = (-1)^{K_{i,j}}(A_j^{(i)}x_{i-1} + b_j^{(i)}),\ K_{i,j} \in \{0, 1\}. \tag{1}$$

We call $(-1)^{K_{i,j}} \in \{+1, -1\}$ the *row sign* of $\eta_{i,j}$. While the key is fixed during the training, all the weights and biases of the model are updated depending on the value of the key.

2.3 Adversary Model

The adversary is a malicious end-user or insider who can download the model architecture and all the parameters from a cloud platform or receive them from the IP owner. The adversary cannot read or probe the key from an instance of the hardware accelerator because it is stored in a tamper-proof memory or a trusted platform module. Despite this, once the model is installed onto the hardware, the adversary can query this *oracle model* with arbitrary inputs a reasonable number of times. It can then observe the logits or the output vector produced by the model.

This adversary model is equivalent to the standard model for logic locking [27, 32]. It is even weaker than the assumptions made in the HPNN paper [7], given that it does not require access to a subset of the training data.

The adversary aims to obtain a correct key, denoted as K^*. When such a key is embedded into the hardware, the encrypted DNN model should be *functionally equivalent* to the original model. Once the adversary acquires such a key, it can replicate and distribute the DNN model without permission from the IP owner. The adversary can also compromise a remote mission-critical system that uses the same DNN model by launching an adversarial attack on the local model.

3 ALGORITHM

3.1 Overview

We propose a systematic attack algorithm for all feasible DNN logic locking schemes we can foresee. It is based on the mathematical framework we established for the security analysis of deep ReLU networks (§3.2).

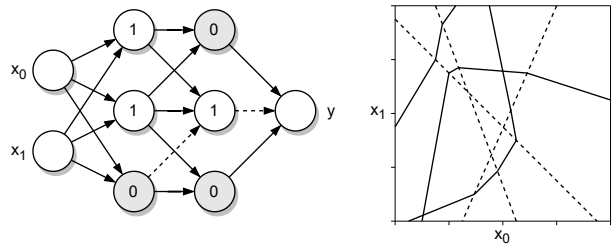
The main algorithm leverages divide-and-conquer to minimize sample and computational complexities (§3.8). Starting from the first hidden layer, it targets one layer in every iteration. For each hidden layer, the algorithm attempts to infer key bits one at a time with an accurate and lightweight algebraic approach (§3.3). When such an attempt fails for some key bits (§3.4), the algorithm initiates a learning-based attack to predict the key values (§3.6). Once all the key values are extracted for a hidden layer, the algorithm executes a rigorous validation and correction procedure to fix potential errors (§3.7). According to our experiments, the attack algorithm can scale to large DNNs and generalize to complex network architectures.

Finally, §3.9 explains how the attack algorithm can be applied to general logic-encrypted DNNs. As such, we conclude that standard logic locking is insecure on DNNs.

3.2 Geometric View of Deep ReLU Networks

We exploit the geometric properties of DNN models to launch our attack. Each neuron induces a $(P-1)$-dimensional *hyperplane* in the P-dimensional input space of that model. The hyperplane comprises points in the input space such that the neuron's corresponding ReLU activation function is at the critical point. Figure 2(b) displays the hyperplanes of a 2-layer DNN model.

The hyperplanes of a DNN model split the input space into disjoint *linear regions*. Given an input example, one can compute the pre-activation values for all neurons of the model through a single forward pass. We use an *activation pattern* vector $m^{(i)} \in \{0, 1\}^{d_i}$ to represent



(a) (b)

Figure 2: (a) Activation statuses of a 2-layer DNN model. The corresponding activation patterns are $m^{(1)} = (1\ 1\ 0)$ and $m^{(2)} = (0\ 1\ 0)$. The dashed path represents a sensitizable path from an intermediate node to the output node. (b) Geometric view of hyperplanes induced by the DNN model in (a). Each dashed line (resp. bent solid line) is induced by a ReLU in the first (resp. second) hidden layer.

the activation statuses for all the neurons in the i-th hidden layer. The j-th element of that vector, $m_j^{(i)}$, is 1 if $A_j^{(i)} x_{i-1} + b_j^{(i)} > 0$, otherwise it is 0. Figure 2(a) shows a 2-layer DNN model and its activation patterns for an input example. Two input examples are within the same linear region if they share the same activation patterns across all hidden layers.

Every linear region is associated with a unique affine transformation from the input space [6]. Given the activation patterns of a linear region, one can recursively compute the weights and the biases of the transformation, layer by layer:

$$M_{j,:}^{(i)} = m_j^{(i)},\ \hat{A}^{(1)} = A^{(1)},\ \hat{b}^{(1)} = b^{(1)}, \tag{2}$$

$$\hat{A}^{(i)} = A^{(i)}(\hat{A}^{(i-1)} * M^{(i-1)}), \tag{3}$$

$$\hat{b}^{(i)} = A^{(i)}(\hat{b}^{(i-1)} * m^{(i-1)}) + b^{(i)}. \tag{4}$$

In the above equations, $*$ denotes element-wise multiplication between matrices, and $M^{(i)}$ is the mask matrix obtained by broadcasting $m^{(i)}$ to all columns of $\hat{A}^{(i)}$. Intuitively, according to the activation patterns, the above recursive formulas select either the inactive or the active region for all ReLU activation functions. We refer to $\hat{A}^{(i)}$ (resp. $\hat{b}^{(i)}$) as the *product weight matrix* (resp. *product bias vector*) of a level-i linear region.

It can be seen that the linear regions exhibit a *hierarchical* structure: a region in one layer depends upon its predecessor regions in the previous layers. Hence, we can derive the following lemma for HPNN:

Lemma 1. *The hyperplane induced by $\eta_{i,j}$ is exclusively determined by $A^{(1)}, \cdots, A^{(i)}$, $b^{(1)}, \cdots, b^{(i)}$ and the row signs in layers 1 through $i-1$.*

Proof. The hyperplane induced by $\eta_{i,j}$ consists of those points x_0 in the input space such that $\pm(A_j^{(i)} x_{i-1} + b_j^{(i)}) = 0$, where $x_{i-1} = (\hat{A}^{(i-1)} x_0 + \hat{b}^{(i-1)}) * m^{(i-1)}$ if $i > 1$. According to the definition of $m^{(i)}$ and Formulas 1-4, x_{i-1} only depends on x_0 itself and the weights, biases and row signs of layers 1 through $i-1$. Notice that the hyperplane does not depend on any row signs in layer i. □

3.3 Key Inference with Basis Vector

Notice that the ReLU function is muted in the inactive region. Let us suppose that an adversary has the capabilities to *i)* find a witness to the hyperplane induced by a designated ReLU function, *ii)* move along a direction from the hyperplane so that the inputs to any other ReLU functions on the same hidden layer remain unchanged, and *iii)* ensure that the designated ReLU function is sensitizable to the output nodes. The adversary can then immediately infer the key bit associated with the designated neuron. In a nutshell, if the output of the DNN model changes accordingly, the designated ReLU function should be moving in the active region. If it remains unchanged, the ReLU stays inactive.

Algorithm 1 The Key Bit Inference Function

1: **Input:** white-box network f, oracle network O, key-protected neuron $\eta_{i,j}$, decrypted key values of preceding layers $K_1^*, \cdots, K_{i-1}^*$
2: **Output:** $K_{i,j}^*$
3: $x^\circ \leftarrow$ `search_critical_point`$(\eta_{i,j})$ ▷ §3.5
4: Observe $m^{(1)}, \cdots, m^{(i-1)}$ with a forward pass from x°
5: Compute $\hat{A}^{(i)}$ according to Formulas 2-3
6: $e_{i,j} \leftarrow$ the j-th standard basis vector in $\mathbb{R}^{d_i}$
7: Find a $v_{i,j}$ using least squares *s.t.* $\hat{A}^{(i)} v_{i,j} = e_{i,j}$
8: **if** $v_{i,j}$ does not exist **then return** $\perp$
9: **if** $O(x^\circ) \neq O(x^\circ + \epsilon \cdot v_{i,j})$ **then return** 0
10: **if** $O(x^\circ) \neq O(x^\circ - \epsilon \cdot v_{i,j})$ **then return** 1
11: **return** $\perp$

Formally, let us consider the pre-activation hidden space of the i-th hidden layer, $\mathbb{R}^{d_i}$. Let $e_{i,j}$ denote the j-th *standard basis vector* of $\mathbb{R}^{d_i}$ and let $z_{i,j}^\circ \in \mathbb{R}^{d_i}$ be a critical point of $\eta_{i,j}$. With a sufficiently small ϵ [6], it is guaranteed that the ϵ-neighborhood of $z_{i,j}^\circ$ does not intersect with any hyperplanes induced by other neurons. In other words, all the points in this neighborhood belong to the same linear region. Moving along $e_{i,j}$ within the neighborhood can only change $z_{i,j}$ but not any other pre-activation values of the same hidden layer because, by definition, $e_{i,j}$ is parallel to the j-th coordinate of $\mathbb{R}^{d_i}$ and orthogonal to any other coordinates. Therefore, we can derive the following lemma:

Lemma 2. *Let $e_{i,j}$ represent the j-th standard basis vector of $\mathbb{R}^{d_i}$, $v_{i,j} \in \mathbb{R}^P$ represent a pre-image of $e_{i,j}$, and $x^\circ(\eta_{i,j}) \in \mathbb{R}^P$ represent a critical point of $\eta_{i,j}$. Then $K_{i,j}^* = 0$ implies that $O(x^\circ(\eta_{i,j}) - \epsilon \cdot v_{i,j}) = O(x^\circ(\eta_{i,j}))$, and $K_{i,j}^* = 1$ implies that $O(x^\circ(\eta_{i,j}) + \epsilon \cdot v_{i,j}) = O(x^\circ(\eta_{i,j}))$.*

Proof. If $K_{i,j}^* = 0$, the pre-activation value $z_{i,j}(x)$ is within the inactive region of the ReLU activation function for both $x_1 = x^\circ(\eta_{i,j})$ and $x_2 = x^\circ(\eta_{i,j}) - \epsilon \cdot v_{i,j}$. Besides, moving along $v_{i,j}$ does not change any other pre-activation values of the same hidden layer, as $e_{i,j}$ is orthogonal to other coordinates in the hidden space. Because all elements of z_i remain unchanged, the oracle network must produce identical outputs for both input samples.

The same reasoning applies to the $K_{i,j}^* = 1$ case. □

Algorithm 1 illustrates how we implement the key bit inference procedure. It starts by finding a critical point x° of the targeted neuron (Line 3). Then it computes the product weight matrix associated with the level-i linear region where x° is located (Line 4-5). Given the product weight matrix $\hat{A}^{(i)}$, we are able to compute the pre-image vector $v_{i,j} \in \mathbb{R}^P$ for $e_{i,j}$. In practice, we find the pre-image with *least squares* (Line 7), which is a built-in function provided by statistics and deep learning frameworks such as SciPy [29] and PyTorch [19]. Finally, the algorithm determines the key bit through queries to the oracle network (Line 9-10). Notice that the statements on Line 9 and Line 10 are contra-positive to their counterparts in Lemma 2. In some rare cases (*e.g.*, when the condition of Lemma 3 does not hold), all the three values obtained from the oracle queries are close to each other. In that circumstance, we attempt to find another x° and start over the entire procedure.

3.4 Sensitization to Input and Output Spaces

In this section, we investigate the sensitization problems surrounding our attack algorithm. First, does a pre-image always exist for every basis vector and every input example? In reality, a DNN can be *expansive* [22] at some specific locations. In a nutshell, if $d_l < d_i$ for some

$l < i$, then $v_{i,j}$ does not exist for every $e_{i,j}$ because $\hat{A}^{(i)}$ is not an onto mapping. Furthermore, inactive neurons reduce the chance of finding a $v_{i,j}$. For a randomly initialized network, about half of the neurons in a hidden layer are inactive for a given input example. This situation worsens when the network encounters a "dying ReLU" problem [15]. Hence, we need a complementary approach to address these issues.

Second, is every hyperplane observable from the primary output nodes? With a little abuse of terminology, we refer to every flat piece of a bent hyperplane as a *boundary*. Intuitively, a boundary may be covered by subsequent layers. The following lemma discusses when a boundary is sensitizable to an output:

Lemma 3. *Consider a set of compatible activation patterns and the corresponding linear region. In addition, consider a boundary of the hyperplane induced by $\eta_{i,j}$ on the linear region. This boundary is sensitizable to an output node y, if there exists a consecutive path from $\eta_{i,j}$ (exclusive) to y such that all neurons along the path are active.*

For instance, in Figure 2(a), the last neuron in the first hidden layer is observable from y because such a consecutive path to y exists from that neuron. In a modern DNN architecture [16], most neurons in hidden layers have multiple successor neurons. Consequently, an intermediate neuron is unlikely to have no sensitizable path to any output nodes.

3.5 Finding Critical Points of a Neuron

Algorithm 1 relies on an essential utility function to find a witness $x^\circ \in \mathcal{X}$ to a designated hyperplane. As discussed in §3.2, a hyperplane generally has $P-1$ dimensions. As a result, a 1-dimensional line is likely to intersect with the hyperplane in the input space at least once [31]. In addition, Lemma 1 states that the hyperplane of a neuron is exclusively determined by the key bits in the preceding layers. Because our algorithm proceeds layer by layer, the current information in the white-box network f is sufficient for our purpose.

The `search_critical_point` algorithm starts by randomly selecting a straight line in the input space. Afterward, it draws random samples along the line and tracks the pre-activation values $z_{i,j}$ of the associated neuron $\eta_{i,j}$. Once it detects two consecutive samples that yield opposite signs for $z_{i,j}$, it performs a standard binary search on the line segment between the two samples. Upon completion, it finds a sample on the designated hyperplane.

3.6 Learning-based Attack

The `key_bit_inference` procedure may not be successful for every key bit due to the problems mentioned in §3.4. In that circumstance, K_i^* contains $\perp$ elements. We perform a supervised `learning_attack` on those remaining bits.

We first convert an HPNN-encrypted network model to a continuous function. Particularly, we substitute every flipping unit with a scalar multiplication operator. This operator multiplies the pre-activation value $z_{i,j}$ with a real number $K'_{i,j} \in [-1, 1]$. We then create a training dataset using the oracle network. Specifically, we randomly generate a set of unlabeled input examples $x_1, \cdots, x_n \in \mathcal{X}$ and query the oracle for the corresponding outputs. During the training process, we fix all the weights and biases of the white-box network. Moreover, for all the key bits in the preceding layers and the non-$\perp$ key bits in the i-th layer, we enforce $K'_{i,j}$ to be -1 (resp. 1) if the original $K_{i,j}$ is 1 (resp. 0). Upon termination, we replace a $\perp$ with a 0 if $K'_{i,j}$ is a positive number and with a 1 otherwise.

However, the learning-based approach cannot guarantee absolute correctness for the extracted key bits. Meanwhile, a single bit of error can devastate our attack on subsequent layers. In this regard, we must validate the correctness of K_i^* before proceeding to the next layer.

3.7 Validating the Correctness of Key Vectors

From Lemma 1, the hyperplane induced by $\eta_{i+1,j}$ is uniquely determined by the key bits in layers 1 through i. Moreover, Lemma 3 proves that hyperplanes are nearly always observable from the output nodes. Leveraging these facts, we can devise an algorithm to check the correctness of K_i^*. Intuitively, if K_i^* is correct, for a level-$(i+1)$ hyperplane of the white-box network, we can almost always find the exact hyperplane of the oracle network at the same location in the input space. On the contrary, if K_i^* is incorrect, it is almost impossible to find any hyperplanes with the oracle network at that location, given the high dimensionality of the input space.

Therefore, the `key_vector_validation` algorithm iterates through all the neurons in the $(i+1)$-th layer. For each neuron, the algorithm searches through the white-box network for a witness x° to the induced hyperplane. It then samples a set of points within a small neighborhood of x°, queries the oracle for the corresponding outputs, and verifies whether all the outputs are on the same linear surface. A hyperplane crossing the neighborhood must exist if those points belong to more than one linear region. To tolerate uncertainties such that *i)* an unrelated hyperplane coincides with the neighborhood, or *ii)* the hyperplane is not sensitizable to the output, a key vector can pass the validation if overlaps of hyperplanes are detected for a majority of the neurons.

Validating the key vector for the last hidden layer requires special treatment, as its next layer (the output layer) has no ReLU activation functions. However, we can directly compare the outputs of the two networks for a set of input examples. This is feasible because all remaining key bits are already determined at the time.

3.8 The DNN Decryption Algorithm

Algorithm 2 The DNN Decryption Algorithm

```
1:  Input: white-box network f, oracle network O
2:  Output: a correct key K*
3:  Let K* be a 2-D array with k rows
4:  for 1 ≤ i ≤ k do                                   ▷ for each hidden layer
5:      for 1 ≤ j ≤ d_i do                             ▷ for each neuron
6:          if η_{i,j} is key-protected then
7:              K*_{i,j} ← key_bit_inference(η_{i,j})   ▷ §3.3
8:      K*_i ← learning_attack(K*_i)                   ▷ §3.6
9:      while not key_vector_validation(K*_i) do       ▷ §3.7
10:         K*_i ← error_correction(K*_i)
11: return K*
```

Algorithm 2 summarizes the main procedure of our attack algorithm. The original decryption problem is successively broken down into the decryption of a single hidden layer to minimize the sample and computational complexities. This strategy is made possible because *i)* all the preceding layers are already decrypted, and *ii)* the decryption of the current layer is independent of the unknown key bits in the subsequent layers.

For every hidden layer, the algorithm traverses all the key-protected neurons and attempts the highly efficient `key_bit_inference` procedure (Line 7). Such an attempt may fail on a subset of the neurons. In that case, the algorithm initiates the `learning_attack` to extract these key bits (Line 8).

The algorithm checks the extracted key values K_i^* before it moves to the next hidden layer (Line 9). If K_i^* cannot pass the validation due to the `learning_attack`, the algorithm enters a heuristic `error_correction` procedure (Line 10). Concretely, it first computes the confidence level for each key bit, defined as the absolute value of $K'_{i,j}$ from the `learning_attack`. A higher confidence level generally implies that the learned key value is more likely to be correct. Throughout

its execution, the procedure maintains a counter of Hamming Distance, whose initial value is 1. In ascending order of the confidence level, the procedure attempts to flip each bit within the limit of Hamming Distance. The counter is incremented by 1 if all attempts fail for the current Hamming Distance.

The next theorem proves the correctness of the DNN decryption algorithm.

Theorem 4. *The DNN decryption algorithm will eventually terminate. Upon termination, it will always return a correct key.*

Proof. *Termination:* The `key_vector_validation` procedure can be executed at most $2^{|K_i|}$ times for the i-th hidden layer, because each time the `error_correction` procedure eliminates one incorrect assignment to K_i. Other procedures can be executed a finite number of times.
Correctness: A row of K^* is confirmed only when it passes the rigorous `key_vector_validation` process. Therefore, the final K^* must be correct. □

3.9 General Security Analysis for DNN Logic Locking

The DNN decryption algorithm can be adapted to defeat a broad category of logic locking schemes on DNN models. As far as we can foresee, an adversary can *a)* choose another arithmetic operator instead of the negation operator; or *b)* modify a single element within the weight or bias matrices instead of the pre-activation value; or *c)* embed key bits to convolutional or max-pooling layers rather than fully connected layers.

All these schemes can be reduced to the standard form that our algorithm can easily handle. For case *a)*, an adversary can propagate the operator to the subsequent layers. For instance, as for a multiplication operator, an adversary can propagate the multiplier to all the successive neurons. Then it can focus on only the fan-out cone in the next hidden layer to extract the associated key bit. On the other hand, for case *b)*, modifying elements of $A_j^{(i)}$ or $b_j^{(i)}$ only changes the geometry of the hyperplane $h_{i,j}$, but not any of $h_{i,k},\ k \neq j$. Similarly, for case *c)*, the modifications only affect the hyperplanes in a local region. Because there is less inter-dependency between the protected neurons in the same hidden layer, an adversary can further leverage a divide-and-conquer strategy to minimize the attack complexity.

4 EXPERIMENTS

4.1 Implementation

We implement the DNN decryption algorithm with PyTorch [19]. We use parallelism to accelerate its computation. For `search_critical_point` and `key_bit_inference`, we initiate multiple instances of these procedures simultaneously for multiple neurons on the same hidden layer. For `error_correction`, we make several guesses of the key vector at a time and execute `key_vector_validation` to verify them in parallel. Besides, we use the built-in Jacobian matrix for any computations related to the product weight matrix $\hat{A}^{(i)}$ to achieve high efficiency.

While the algorithms described in §3 assume a sequential DNN architecture, it can be easily adapted to accommodate more diverse network topologies, including residual connections and attention units. Generally speaking, we process the hidden layers in a topological order. If a hidden layer has multiple preceding layers, all key bits in those layers must be processed beforehand.

As for the `learning_attack`, we substitute each $K'_{i,j}$ with a *sigmoid* function. The sigmoid function has a $[-1, 1]$ range, which coincides with that of a key bit. We choose the mean squared error between the output vectors of the encrypted network and the oracle network as the loss function. During training, we periodically settle down those key bits that have reached the confidence threshold.

4.2 Experimental Setup

We assess the proposed algorithm on the following DNN architectures: *a) MLP*, a multilayer perceptron with 2 hidden layers; *b) LeNet*, a ReLU variant of LeNet-5 [14]; *c) ResNet*, the 18-layer version of Residue Network [12]; and *d) V-Transformer*, a ReLU variant of Vision Transformer [8] (ViT) with 12 self-attention blocks. We conduct all experiments on a Linux workstation with a 2.4GHz CPU and an Nvidia RTX A6000 graphics card.

We apply HPNN to encrypt the aforementioned DNNs. Given a specific key size, we *i)* equally distribute the key bits to all designated hidden layers, *ii)* embed key bits to a set of neurons randomly selected from every hidden layer, and *iii)* assign a value for every key bit uniformly at random. After that, we train the DNN models as functions of the keys until they converge. We launch two types of attacks on the resulting DNN models: *a)* a monolithic learning-based attack, which only applies the method described in §3.6; and *b)* the proposed comprehensive DNN decryption algorithm (Algorithm 2).

We use four metrics to measure the effectiveness and efficiency of the attacks: *a) accuracy*, which is the percentage of correct predictions on the testing dataset; *b) fidelity* [13], which is the percentage of exactly recovered key bits; *c) execution time*; and *d) query complexity*, which is the total number of queries made to the oracle network. Extracting a model with high accuracy could facilitate IP piracy, whereas extracting a model with high fidelity also enables adversarial attack. Table 1 shows our evaluation results. To compute the baseline accuracy, we randomly generate 16 incorrect keys for every network model and then compute the average of their accuracy.

4.3 The Monolithic Learning-based Attack

For every network model, this attack first generates a set of input examples and then queries the oracle network for corresponding output vectors. It terminates when either *i)* all key bits have reached the confidence threshold, or *ii)* neither the accuracy nor the fidelity increases for a number of consecutive epochs.

Table 1 shows that the learning-based attack is effective for smaller networks with small key sizes, but becomes less effective as the key size grows. For larger networks, the learning attack alone cannot achieve high fidelity regardless of the key size. We tried various hyperparameter settings and got similar results. Hence, we resort to the DNN decryption algorithm for large key sizes.

4.4 The DNN Decryption Attack

With an orchestration of the `key_bit_inference`, `learning_attack`, `key_vector_validation` and `error_correction` procedures, our proposed algorithm achieves 100% fidelity in all instances.

Figure 3 breaks down the total execution time among these procedures. The percentage of the time consumed by each procedure is related to the network architecture. Since the *MLP* network is highly contractive (784 input nodes and 256/64 neurons in the first/second fully connected layer), `key_bit_inference` alone can easily decrypt all three instances. Only a negligible amount of time is spent on `key_vector_validation`. In contrast, all the remaining network architectures are expansive at some locations. For *LeNet*, `learning_attack` can recover almost all key bits. The remaining key bits are recovered by `key_vector_validation` and `error_correction` in a small amount of time. For *ResNet* and *V-Transformer*, `learning_attack` is less effective due to larger model capacities. Therefore, most of

Table 1: Experiment results of attacks against logic locking on DNNs.

DNN *(Dataset)*	Key *(bits)*	Original Accuracy	Baseline Accuracy	Monolithic Learning-based Attack				DNN Decryption Attack			
				Accuracy	Fidelity	Time *(s)*	# Queries	Accuracy	Fidelity	Time *(s)*	# Queries
MLP *(MNIST)*	32	98.1%	27.6%	98.1%	100.0%	2.70	1,000	98.1%	100.0%	0.18	156
	64	98.1%	10.4%	78.6%	93.8%	5.26	1,000	98.1%	100.0%	0.25	252
	128	98.1%	7.5%	58.5%	87.6%	5.31	1,000	98.1%	100.0%	0.35	444
LeNet *(MNIST)*	32	99.0%	86.7%	99.0%	100.0%	9.16	2,000	99.0%	100.0%	6.48	1,166
	64	99.0%	49.9%	98.9%	96.9%	16.46	2,000	99.0%	100.0%	7.49	1,262
	128	99.0%	16.4%	88.6%	85.9%	16.51	2,000	99.0%	100.0%	10.02	1,464
ResNet *(CIFAR-10)*	64	95.2%	95.2%	95.2%	57.8%	1,138.69	10,000	95.2%	100.0%	256.16	17,296
	128	95.2%	95.1%	95.2%	58.6%	1,308.11	10,000	95.2%	100.0%	346.54	28,096
	196	95.2%	94.9%	95.1%	56.6%	1,807.58	10,000	95.2%	100.0%	1,246.71	122,744
V-Transformer *(CIFAR-10)*	64	98.9%	98.7%	98.8%	56.2%	6,830.77	10,000	98.9%	100.0%	954.27	12,864
	128	98.9%	98.4%	98.7%	54.7%	8,250.18	10,000	98.9%	100.0%	1,444.86	18,912
	196	98.9%	98.0%	98.7%	54.6%	7,855.54	10,000	98.9%	100.0%	6,705.39	95,808

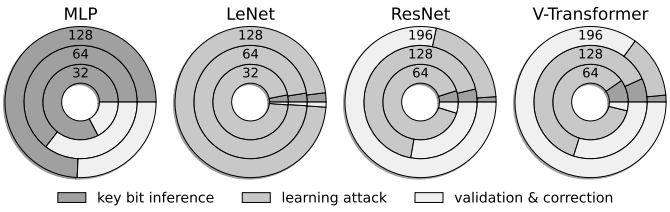


Figure 3: Breakdown of execution time among procedures for different network architectures and key sizes.

the execution time is spent on `key_vector_validation` and `error_correction`.

5 RELATED WORK

5.1 I/O Attacks on Logic Locking

The SAT attack [27] and its SMT variant [4] are the most prevalent and potent I/O attacks against logic locking. In every iteration, it *i)* calls a SAT solver for a distinguishing input pattern, *ii)* queries the oracle circuit for the corresponding output pattern, and *iii)* adds the I/O pair and a fresh copy of the circuit to the SAT solver as a new constraint. The attack terminates when no more distinguishing input patterns exist, and a correct key can then be extracted from the SAT solver. At a glance, the same method can be applied to the DNN decryption problem. Unfortunately, a deep ReLU network cannot be encoded as Boolean formulas but rather has to be encoded as mixed-integer linear equalities. Due to this gap, attacking a locked DNN model monolithically is not computationally feasible.

5.2 Reverse Engineering Deep ReLU Networks

Differential attacks [6, 13, 23] consistently query an oracle network to recover the weights and biases of the original network. While these studies provide valuable insights into deep ReLU networks, even the state-of-the-art implementation [6] requires more than 2^{21} queries to partially reconstruct a DNN with less than 1,000 neurons. They are also struggling with DNNs that have more than three hidden layers.

6 CONCLUSION

This paper presents the first attack on logic-encrypted DNNs using formal techniques. It combines algebraic and learning-based approaches to extract a correct key of a victim DNN. The experimental results indicate that it can scale to large DNNs and complex network architectures. Our findings suggest that binary key bits are vulnerable when embedded in deep ReLU networks. We believe they should be embedded in the computational units within AI accelerators, thus enabling more correlations between key bits and more significant impacts on the final outputs. We leave this problem to our future research.

REFERENCES

[1] Yossi Adi et al. 2018. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *USENIX Security*. 1615–1631.
[2] Manaar Alam et al. 2022. Nn-lock: A lightweight authorization to prevent ip threats of deep learning models. *ACM JETC* 18, 3 (2022), 1–19.
[3] Ross Anderson. 2020. *Physical Tamper Resistance*. 483–521 pages.
[4] Kimia Zamiri Azar et al. 2019. SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks. *CHES*, 97–122.
[5] Battista Biggio et al. 2013. Evasion attacks against machine learning at test time. In *ECML PKDD*. Springer, 387–402.
[6] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. 2020. Cryptanalytic extraction of neural network models. In *CRYPTO*. 189–218.
[7] Abhishek Chakraborty, Ankit Mondai, and Ankur Srivastava. 2020. Hardware-assisted intellectual property protection of deep learning models. In *DAC*. 1–6.
[8] Alexey Dosovitskiy et al. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*.
[9] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS*. 1322–1333.
[10] Brunno F Goldstein et al. 2021. Preventing DNN model IP theft via hardware obfuscation. *IEEE JETCAS* (2021), 267–277.
[11] Jia Guo and Miodrag Potkonjak. 2018. Watermarking deep neural networks for embedded systems. In *ICCAD*. 1–8.
[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*.
[13] Matthew Jagielski et al. 2020. High accuracy and high fidelity extraction of neural networks. In *USENIX Security*. 1345–1362.
[14] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[15] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. 2019. Dying relu and initialization: Theory and numerical examples. *arXiv:1903.06733* (2019).
[16] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. 2017. The expressive power of neural networks: A view from the width. In *NIPS*.
[17] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *ICML*. 807–814.
[18] Brooks Olney and Robert Karam. 2022. Protecting Deep Neural Network Intellectual Property with Architecture-Agnostic Input Obfuscation. In *GLSVLSI*. 111–115.
[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*.
[20] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. 2006. vTPM: virtualizing the trusted platform module. In *USENIX Security*. 305–320.
[21] Prajit Ramachandran, Barret Zoph, and Quoc V Le. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941* (2017).
[22] Stefano Recanatesi et al. 2019. Dimensionality compression and expansion in deep neural networks. *arXiv:1906.00443* (2019).
[23] David Rolnick and Konrad Kording. 2020. Reverse-engineering deep relu networks. In *ICML*. 8178–8187.
[24] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. 2008. EPIC: Ending piracy of integrated circuits. In *DATE*. 1069–1074.
[25] Maria Sommerhalder. 2023. Hardware Security Module. *Trends in Data Protection and Encryption Technologies* (2023), 83–87.
[26] Jacob Steinhardt, Pang Wei W Koh, and Percy S Liang. 2017. Certified defenses for data poisoning attacks. In *NIPS*.
[27] Pramod Subramanyan, Sayak Ray, and Sharad Malik. 2015. Evaluating the security of logic encryption algorithms. In *HOST*. 137–143.
[28] Ashish Vaswani et al. 2017. Attention is all you need. In *NIPS*.
[29] Pauli Virtanen et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.
[30] Mingfu Xue et al. 2021. Intellectual property protection for deep learning models: Taxonomy, methods, attacks, and evaluations. *IEEE TAI* 3, 6 (2021), 908–923.
[31] Xiaodong Yang et al. 2021. Reachability analysis of deep ReLU neural networks using facet-vertex incidence.. In *HSCC*. 19–21.
[32] Hai Zhou, Amin Rezaei, and Yuanqi Shen. 2019. Resolving the trilemma in logic encryption. In *ICCAD*. 1–8.