DNNCloak: Secure DNN Models Against Memory Side-channel Based Reverse Engineering Attacks

Yuezhi Che
yche3@hawk.iit.edu
Illinois Institute of Technology
Chicago, USA

Rujia Wang
rwang67@iit.edu
Illinois Institute of Technology
Chicago, USA

2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00023

***Abstract*—As deep neural networks (DNN) expand their attention into various domains and the high cost of training a model, the structure of a DNN model has become a valuable intellectual property and needs to be protected. However, reversing DNN models by exploiting side-channel leakage has been demonstrated in various ways. Even if the model is encrypted and the processing hardware units are trusted, the attacker can still extract the model's structure and critical parameters through side channels, potentially posing significant commercial risks. In this paper, we begin by analyzing representative memory side-channel attacks on DNN models and identifying the primary causes of leakage. We also find that the full encryption used to protect model parameters could add extensive overhead. Based on our observations, we propose DNNCloak, a lightweight and secure framework aiming at mitigating reverse engineering attacks on common DNN architectures. DNNCloak includes a set of obfuscation schemes that increase the difficulty of reverse-engineering the DNN structure. Additionally, DNNCloak reduces the overhead of full weights encryption with an efficient matrix permutation scheme, resulting in reduced memory access time and enhanced security against retraining attacks on the model parameters. At last, we show how DNNCloak can defend DNN models from side-channel attacks effectively, with minimal performance overhead.**

I. Introduction

In recent years, deep neural networks (DNNs) have become increasingly popular as a result of their superior accuracy in a wide range of machine learning applications, including automatic speech recognition [30], image recognition[27], and AI for games [26]. On the other hand, training a successful model is expensive since it requires a huge amount of labeled data and hardware resources. For example, the collection of over 1.4 million hand-annotated images and 20 thousand categories in ImageNet [25] requires considerable effort in terms of manpower and material resources. Therefore, the knowledge of a well-trained DNN model is intellectual property (IP) and should be kept confidential. Moreover, knowing of the DNN model, such as its structure or parameters, can increase the success rate of adversarial attacks [15, 20], which becomes one of the major security concerns associated with the use of DNNs [9]. The model parameters, including the weights and intermediate results, can be hidden by using the existing encryption, like direct encryption and counter mode encryption. Although encryption causes performance degradation, it is necessary for security. However, the increasing commercial value of DNN comes with more potential risks of attacks. Previous works show that a knowledgeable attacker is able to conduct an attack to extract the hidden information of the model even if the data is encrypted through side-channels, such as access patterns [9, 10], the timing difference in the cache hierarchy [8, 17, 31]. Among all these side-channel attacks, from the architectural perspective, it is relatively easy to observe and attack through the memory access patterns [9, 10], including the access type, address, frequency, and memory footprint. The attacker only needs to probe the external memory bus and record the signals on the command and address buses [12], which cannot be protected by using data content encryption. Therefore, in this work, we focus on defending the reverse engineering attacks by memory access pattern side-channel.

The memory accesses during DNN inference exhibit very strong deterministic patterns, which can reveal the structure of the DNN model. Once the structure of the DNN model is known, the attacker can then conduct other attacks, such as the model extraction [29] and the membership inference attack [18], for further model stealing. During DNN inference, the working layer always reads the preceding layer's output as its input feature maps. If the feature maps and weights are stored in off-chip memory, the memory access patterns can easily expose the layer boundary between two consecutive layers by detecting read-after-write (RAW) dependency [10]. Additionally, the memory footprint exposes the dimension of each layer directly, enabling for the leakage of the overall DNN structure through the access pattern side-channel.

While access pattern leakage poses a significant threat to DNN model IP, there is a lack of lightweight and effective defense mechanisms. The general method to defend memory access pattern leakage is Oblivious RAM (ORAM) [7], which is a provable secure cryptographic protocol. ORAM turns memory accesses into indistinguishable access sequences by address remapping and adding redundant dummy accesses. However, the state-of-the-art ORAM protocols and optimizations [2, 3, 24, 28] still show a very high protection overhead for most applications. Except for ORAMs, recent studies propose shuffle-based [16] and software-based [13] obfuscation schemes to protect DNN model from access pattern leakage. However, the naive shuffle-based obfuscation[16] is not able to hide the accesses with Read-After-Write (RAW) dependency; the software-based scheme[13] generates a new model with the same model accuracy as the original, but the newly generated

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00023

model is still exposed and becomes a new target for attacks.

In the paper, we propose DNNCloak, a lightweight and secure DNN framework against memory side-channel attacks, which protects both DNN model structure and parameters. DNNCloak is the first architecture-level obfuscation design to protect DNN against side-channel attacks. This work makes the following contributions:

- We observe that most existing side-channel attacks on DNN only make use of partial access patterns. Not all access patterns are necessary to be hidden; we only need to protect the access patterns that leak critical hints about the DNN model structure. Based on our observations, we summarize four critical leakage hints that are actually used in reverse engineering.
- Our proposed DNNCloak provides three schemes, *Layer Divider*, *Layer Shrinker*, and *Layer Obfuscator*, to hide the critical leakage hints, so protecting the model structure efficiently.
- Our proposed DNNCloak provides a novel structure called Random Permutation Matrix to protect the model parameters. We demonstrate that this method is secure and more efficient compared to the full encryption or existing partial encryption scheme.
- Our results show that DNNCloak may take down memory cycles to 86% of the unprotected baseline model, while achieving high security protection.

II. Background

A. DNN Basics

*1) **DNN Architectures**:* A typical DNN architecture is shown in Figure 1(a). There are several types of hidden layers between the input and output layers: convolutional layers, pooling layers and fully connected layers. A convolutional layer applies a convolutional operation to the input feature maps (IFMs) and passes the output feature maps (OFMs) to the next layer. During every convolution, a filter with weights slides over the IFMs and transforms the input matrix into a new matrix with extracted convoluted features. A pooling layer is applied over feature map to reduce the spatial size of the representation, thus reducing the number of parameters and computation in the network. A fully connected layer connects all the inputs from the previous layer to every activation unit of the next layer, which is widely used as the last few layers of a DNN.

*2) **DNN Accelerators**:* Since the operations in DNN, such as the convolution operation, can be naturally calculated in parallel, the DNN accelerator can exploit parallelism by utilizing direct connections between an array of processing engines (PEs). As shown in figure 1 (b), a typical DNN accelerator primarily consists of an array of PEs and a global buffer. During the DNN inference, the IFMs and filters are partitioned into small tiles and then processed by the accelerator. The global buffer holds the intermediate results within the same layer. After the computing, the accelerator combines the results and writes the OFMs to the external memory [4, 10]. Therefore, in the DNN accelerator, the feature maps and weights are stored

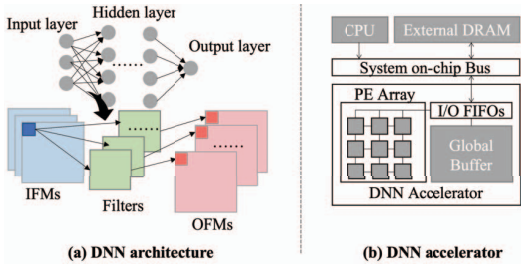


Fig. 1: DNN architecture and a typical DNN accelerator.

in the *external memory*, while the intermediate results of small tiles in the same layer are kept in the on-chip global buffer. The memory traffic could expose the complete access pattern between layers in this situation, so the attack steps will be discussed in the next section.

B. Reverse DNN via Memory Access Patterns

In this section, we introduce ReverseCNN [10], a general memory access pattern side-channel attack framework.

*1) **Threat Model**:* When using the DNN accelerator, the DNN computations are offloaded from the host CPU to the DNN accelerator. As discussed in Section II-A2, the on-chip memory size of the accelerator is limited, so the majority of data, including the IFMs, weights, and OFMs, are transmitted on the external memory bus between the DNN accelerator and memory [10]. In such a use case, we assume that all the computation units, including the accelerator and the host CPU, are trusted. Also, the communication between the host CPU and the accelerator is secured with encryption, so the data content is not exposed to the attacker. However, the memory access pattern to the external memory, including the read and write type, intensities, and frequencies, can be observed. During the DNN inference, the accelerator first accesses the off-chip memory for weights and IFMs. After each matrix computation, it writes the OFMs back to the memory. As a result, the attacker can directly obtain the memory access patterns by snooping the address bus.

*2) **Attack Steps**:* To reconstruct the structure of the DNN model through access patterns, the attacker must get the layer sequence and dimensions. We summarized the attack steps in ReverseCNN[10], which is followed by other similar attacks [9]. The first step to *Identify the number of layers.* The memory access patterns may expose the layer boundaries through read-after-write (RAW) dependency. We show an example of memory access patterns during DNN inference in Figure 2, similar to the ones identified in [10], the RAW dependency of feature map accesses can clearly expose the layer boundaries. At the end of each convolutional layer, the output of a preceding layer will be read as the input by the following layer. The second step is to *identify the dimension of each layer.* The attacker can easily infer the dimensions of the feature maps and filters according to their accessed memory footprint within the same layer. First, the filter is read-only and the access patterns of the feature maps have the RAW dependency. Second, the same type of data is stored in a continuous memory space, so the attacker can easily distinguish the feature maps and filters

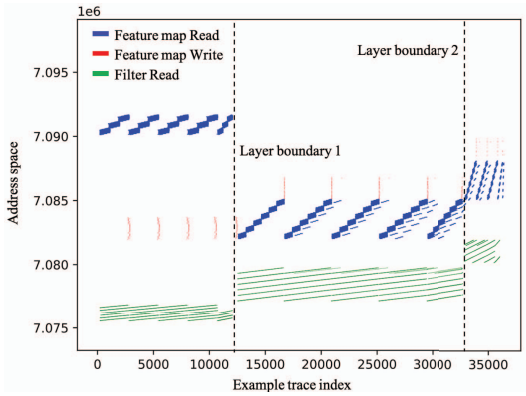


Fig. 2: The exposed memory access pattern trace during DNN inference.

and obtain their dimension in each layer. In last step, the attacker could *reverse the weights* by exploring the vulnerability introduced by zero pruning techniques. Zero pruning has been recently implemented in multiple DNN accelerators [21] due to its efficiency. The zero pruning only reads and writes non-zero values in the weight matrices, so the attackers can effectively know the value of pixel in OFM becomes zero ($0 = f(\sum_i w_i x_i + b)$). The attackers can then get a linear equations for w_i and b with known x_i, so weights can be expressed as a function of the bias. In ReverseCNN[10], attackers may fully recover weights by carefully modifying the non-zero value in the input.

C. DNN Weights Encryption

Memory encryption, such as direct encryption and counter mode encryption, is used to protect the confidentiality of DNN weights stored off-chip. The direct encryption uses the AES encryption engine and applies the same global key to all data, and has high decryption latency; while the counter model encryption [11] uses a counter-based AES encryption engine to generate a one-time pad (OTP), encryption and decryption are accomplished by simply XORing the OTP with the target data. OTPs are never reused, so counter mode encryption provides a higher security level than direct encryption. We use counter model encryption as memory encryption. However, full memory encryption still reduce over 50% of overall performance due to the low hardware bandwidth [32]. So, **partial encryption** is proposed, for example, the *criticality-aware smart encryption* scheme [32] selectively encrypts 40% of the total parameters and achieves the same security guarantee as full encryption, improving performance by saving encryption time. Although data can be protected by memory encryption, the exposed access pattern between layers still reveal sensitive information, as discussed above (§II-B).

III. Observations and Challenges

A. Critical Leakage Hints through Access Patterns

By analyzing the attack steps (§II-B2), our key insight is that **we do not need to obfuscate all memory accesses during DNN inference to protect the DNN structure**, since not all access patterns information is useful for attackers. We refer to these useful hints that potentially reveal the DNN structure as *critical leakage hints*, which need to be hidden and protected. We list four critical leakage hints: (1) *RAW (read-after-write) dependency on feature maps* and (2) *the memory footprint accessed in each layer*; these two hints are introduced (§II-B2), which expose the layer boundaries and dimensions, respectively. Besides, we find out two more hints: (3) *Memory access intensities* and (4) *The accessed memory address space of filters*. The access intensities represent the memory access frequency within a periodic time, and it can be exploited through timing side-channel [6]. The access intensities of the Conv and Pool layer are very different: in our tests, the Conv layers (95.25 accesses/ms) have a greater access intensity than the Pooling layers (31.52 accesses/ms) on average, so the layer boundaries between different type of layers may be exposed through access intensities. *The accessed memory address space of filters* also can expose the layer boundary. The filters are allocated in consecutive memory space; as shown in figure 2, there are noticeable deviations at each layer boundary since a new layer needs access to a new set of filters.

B. Challenges

1) RAW dependency cannot be hidden by shuffling: Although we have identified the critical leakage hints, hiding them is still challenging. For example, the DNN access pattern inherently has the RAW dependency due to the data transfer between adjacent layers. Therefore, the attacker can always observe a large amount of accesses with RAW dependency during DNN inference. The challenge is that the RAW dependency cannot be hidden by shuffling the mapping between on-chip addresses and physical addresses. Moreover, adding some fake RAW accesses for obfuscation may not be sufficient, since attacker can still determine the difference by examining a large number of original access sequences with true RAW dependency, unless the added fake RAW accesses are large enough to completely obfuscate the original ones, which would be extremely costly.

2) Limitations of Existing Defense Schemes: In order to completely eliminate access pattern leakage, ORAM[7] was proposed and has been optimized in past decades [24, 28]. ORAM is a general provable secure protocol designed for obfuscating all memory accesses; however, it is less desirable to applying ORAM for DNN applications, because of the significant performance overhead with random memory reads and writes. Liu et al. [16] proposed a shuffle-based obfuscation scheme for DNNs, aiming to reduce the overhead of ORAM-based approach. However, the RAW dependency could not be hidden. Besides, tracking the entire shuffling needs a huge mapping table, which is impossible to store on-chip. Li et al. [13] proposed NeurObfuscator, a full-stack software solution that provides obfuscation toward the layer sequence of the network and the dimension of layers, and can hide the original access patterns. However, the authors claim that their software-based obfuscation has no affect on the model's functionality, which means that the attacker may still reverse the obfuscated

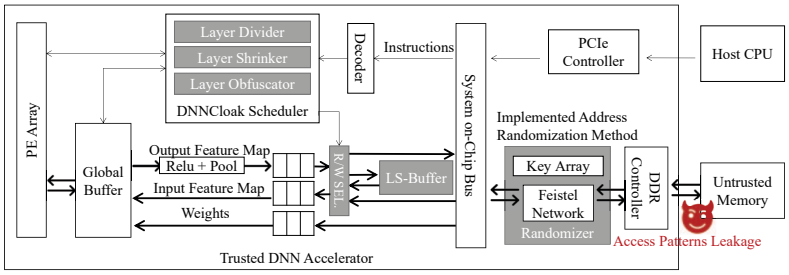


Fig. 3: The high-level overview of DNNCloak architecture.

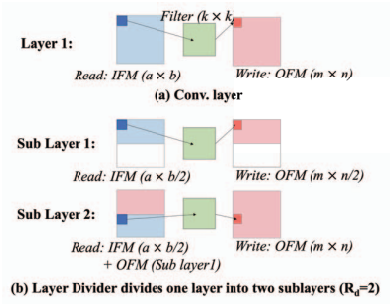


Fig. 4: The example of Layer Divider.

DNN model through a new set of memory access patterns. Since the obfuscated model is functionally identical to the original, it could serve as a new target for attackers.

IV. The Design of DNNCloak Framework

A. DNNCloak Architecture Overview

We propose DNNCloak, a lightweight defense framework for DNN models under reverse engineering attacks. Instead of obfuscating all access patterns, we focus on efficiently hiding the critical leakage hints to protect the DNN model. Figure 3 shows a high-level overview of DNNCloak architecture design in a typical DNN accelerator architecture. Note that our proposed parts are highlighted in grey. DNNCloak architecture consists of three schemes (*Layer Divider*, *Layer Shrinker*, and *Layer Obfuscator*), which obfuscate the data flow between the accelerator and the external memory. The R/W Selector and LS-Buffer are added to assist the Layer Shrinker method. The address remapping function unit (Randomizer) utilizes Feistel Network, which provides a lightweight random address mapping [19, 22, 23]. Briefly, it is an XOR-based algorithm with the requirements of a secret key array, and it provides a one-to-one mapping with low overhead, 6B on-chip size and less than 3 cycles latency [23].

B. DNN Model Structure Protection Mechanisms

DNNCloak includes three schemes to hide the critical hints. Among these four hints introduced in section III-A, hint 4 can be concealed by address randomization, the remaining hints are obfuscated by the DNNCloak schemes.

1) ***Layer Divider (LD):*** Considering the challenges (§III-A) for hiding the hint 1 RAW dependency, the idea of Layer Divider is to divide one single Conv layer into multiple smaller sublayers, thereby increasing the total number of observed layers. For example, Figure 4 (a) shows one Conv layer with a $a*b$ IFM, a $k*k$ filter, and a $m*n$ OFM. Figure 4 (b) shows an example of Layer Divider, the original layer is divided into two sub-layers. DNNCloak reads half of the input $a*b/2$ as the IFM for the first sub-layer. Then write the results as the first sub-layer's OFM and the size is $m*n/2$. Next, the second sub-layer reads the other half of the original input $a*b/2$ and also reads the preceding layer's OFM, and only the half original input does the multiply-accumulate operations. Each sub-layer is performed continuously with the same filter so we can keep the filter on-chip to avoid unnecessary off-chip memory accesses. Finally, we can get a complete results by concatenating the results of the two sub-layers.

While we divided the layer into two sub-layers in the preceding example, for security purposes, the Layer Divider should randomly divide each layer into different number of sub-layers. We define a divide rate R_d that represents the average number of sub-layers that each layer is divided. By adjusting the Layer Divider's R_d, we can arbitrarily increase the number of original layer boundaries. While the increased R_d causes more frequent division of the original layer and improves security, it introduces extra read and write operations. As a result, the attacker will observe more layer boundaries than the actual ones from the RAW dependency.

2) ***Layer Shrinker (LS):*** While the Layer Divider scheme obfuscates the layer sequence, the critical leakage hint 2 remains, which exposes the dimension of each layer through the memory footprint. We then propose the Layer Shrinker to address this issue. The main idea of Layer Shrinker is to reduce the exposed memory footprint size by buffering a portion of the feature maps on-chip. Figure 5 depicts the Layer Shrinker organization and shows two different scenarios. Three components are included within the Layer Shrinker: the LS-Controller, the LS-Map, and the Threshold Generator. In addition, we have an R/W Selector and LS-buffer on-chip for the method process. The LS-Buffer can store OFMs and directly provide them to the following layer as IFMs, and the R/W Selector manages the data flow.

Figure 5 (a) shows when the DNN accelerator writes the OFMs. ① LS-Controller receives the instructions that indicate the memory address for writing OFMs. And the OFM instructions could imply the starting memory address a_0 and the OFMs size S_{OFM}. ② Instead of writing all of the OFMs to memory, we can buffer a portion of OFMs on-chip. A constant buffer size S_{buffer} may be easily reversed by attackers, so we hide the actual buffered size, $S_{buffer} \times T$, with a random T generated by the Threshold Generator. ③ Then, LS-Controller updates the LS-Map, which records the information about buffered addresses, such as the starting address a_0 and the corresponding T. ④ Lastly, LS-Controller controls the R/W Selector to buffer a portion of the data ($S_{buffer} \times T$) and write the rest ($S_{OFM} - S_{buffer} \times T$) to memory.

Figure 5 (b) shows reading the IFMs. ① LS-Controller receives the instructions indicating the memory addresses needed to be read. ② Next, LS-Controller check the LS-

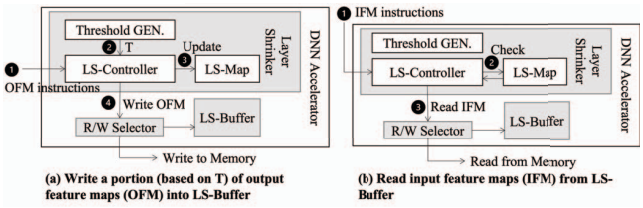


Fig. 5: Layer Shrinker design illustration.

Map to get the information about a_0 and T. So the needed memory address is start from the $a_0 + S_{buffer} \times T$ with size $S_{OFM} - S_{buffer} \times T$. ③ LS-Controller manages the R/W Selector to control the data flow, reading from the LS-Buffer and memory.

*3) **Layer Obfuscator (LOB)**:* Hint 3 leaks information through the access intensities between different types of layers. To obfuscate the different access intensities and hide the exposed layer boundary between the different types of layers, we propose the Layer Obfuscator. The pooling layers have lower access intensities compared to the Conv layers, so the idea of Layer Obfuscator is to add dummy accesses to make the access intensities the same. Since the pooling layer takes a small part of DNN execution time, the extra overhead has little effect to the overall performance. Additionally, we used an address randomization method that disrupts the initial spatial locality, rendering the added dummy accesses indistinguishable.

C. DNN Model Weights Protection

After obtaining the model structure, the attacker can continue to attack the model and reverse the weights by exploiting the vulnerabilities caused by the conventional zero pruning technique (§II-B2). As described in section II-C, full memory encryption is able to protect the model parameters by making zero and non-zero values indistinguishable to attackers, but it is costly due to encryption and decryption latency. Partial encryption can enhance performance, but the improvement depends on the ratio of partial encryption. If partial encryption ratio is insufficient for the sake of reducing encryption latency, the model's ability to resist retraining attacks will be diminished. We show our related observations in section IV-C2.

*1) **Random Permutation Matrix (RPM)** Format :* To reduce the latency caused by memory encryption, we propose a novel scheme, *Random Permutation Matrix (RPM)*, which encrypts just a small portion of the data, such as some metadata, to achieve the same level of protection to the model as full encryption.

Figure 6 shows an example of using RPM data format. Note that $n0$ refers to those non-critical weights with very small values, also can be regarded as zero. The position of each element in the matrix is randomly permuted, and a bit map is added as metadata at the head of the data block. There are two components in the bit map, the position bits ($B_{position}$) and the order bits (B_{order}). For example in figure 6, if the matrix size is 3×3, such as all filters in VGG16, then the $B_{position}$ should be 9 bits, which indicates the position of the $n0$. So in this example, $B_{position}$ is 110110101. And the B_{order} is

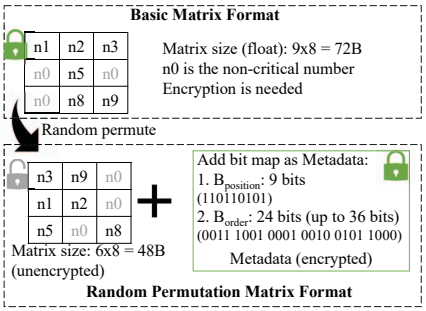


Fig. 6: Random Permutation Matrix data format.

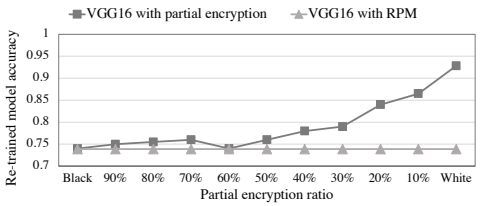
24 bits (up to 36 bits), which records the original position and is used to restore the matrix order. So, for 3×3 matrix such as VGG16, the total size of the metadata is less than 5.7B. Generally, to record a permuted $m \times m$ matrix, it takes $log_2(m \times m)$ bits for each position. So, the increasing size of the matrix would increase the overhead of both $B_{position}$ and B_{order}. When $m = 3$, the size ratio of metadata and matrix data is 0.078; while when $m = 11$, the ratio is increased to 0.125. Note that most filter size is less than 11×11. Therefore, the overhead of the metadata is 7.8% to 12.5% in common.

With RPM, we only need to encrypt the metadata $B_{position}$ and B_{order} to achieve the same security and accuracy of full weights encryption. The possibility of restoring the order of a $m \times m$ matrix is $\frac{1}{m!}$, and that is for only one matrix in one convolution. Therefore, reordering the permuted filters is impossible.

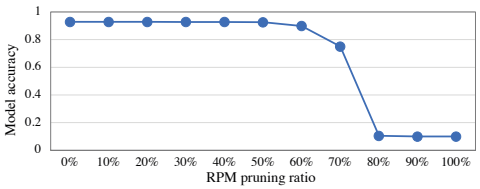
Moreover, as discussed, the $n0$ in figure 6 refers to the non-critical weights, and can be pruned as zero. So, the more $n0$, the less overhead of metadata. Notably, we also do not need to store $n0$ in the data block since the metadata indicates the position of each $n0$. Hence, if the matrix is 3×3, the matrix size would be 72B before permuting as shown in figure 6, which requires two data blocks to store one matrix. After random permutation, if there are more than two $n0$ in the matrix, then the matrix size is less than 64B even with the metadata so that only one data block is fetched from memory each time. Therefore, RPM with pruning can significantly reduce memory access time.

2) RPM Discussions: We discuss the security of RPM by comparing it with full encryption and partial encryption in terms of retraining model accuracy.

Observation 1: RPM can achieve the same security level of full encryption with much less overhead. We compare RPM with partial encryption scheme SEALing [32] to show the model's ability to resist retraining attacks. We use the same settings as [32]: using the VGG16 model as an example, and the dataset is CIFAR-10; assume the retraining dataset contains only 10% (5000 images) of the original training dataset, and augment the retraining dataset to 45000 images; we first select the matrix with the largest sum to encrypt. Then, as shown in figure 7 (a), when the selective encryption ratio is above 50%, the partial encryption achieves the same security level as the full encryption; but when the encryption ratio is insufficient, for



(a) Re-trained VGG16 accuracy comparison between partial encryption and RPM



(b) VGG16 accuracy after pruning different ratios of weights

Fig. 7: Observations on Random Permutation Matrix design.

example, when the ratio is 20%, the model can be retrained to over 85% accuracy. On the other hand, without knowing the bit map, the model with RPM always has the same security level as full encryption during retraining tests.

Observation 2: RPM can work with pruning to reduce overhead without hurting inference accuracy. In our experiments, we prune the smallest weights to zero in VGG16, and with different pruning ratios, the model accuracy is shown in figure 7 (b). When 40% of the smallest weights are pruned to zero, the model accuracy only decreases less than 0.1%; when the pruning ratio is 50%, the model accuracy decreases 0.2%; when the pruning ratio is 60% or greater, the model accuracy degrades rapidly.

In summary, RPM has the same security level as full encryption for model IP protection. And it provides strong security against retraining attacks and, under the same retraining conditions, achieves the same model accuracy as full encryption or 60% partial encryption, while we only need to encrypt the bit map, so the encryption ratio is about 10%. Therefore, RPM reduces the encryption latency significantly; in addition, RPM can reduce the memory access time since the non-critical weights can be pruned without leaking any information. We show the performance results in section VI-B.

V. DNNCloak Design and Security Discussions

In this section, we discuss the following design choices and their impact on defense mechanisms: the choice of divide rate R_d (§IV-B1); the choice of LS-Buffer size S_{buffer} (§IV-B2); and how DNNCloak can protect against reverse engineering attacks on the DNN model. Figure 8 illustrates DNNCloak schemes step-by-step. To start with, Figure 8 (a) shows a sketch of a basic DNN access pattern as an example. The X-axis is the trace index which indicates the processing timeline, and the Y-axis is the memory address space. The first RAW reveals the layer boundary and also the memory footprint exposes the feature map size Z_i^{out} and filters size Z_i^f (i is the current layer index) as introduced in section II-B.

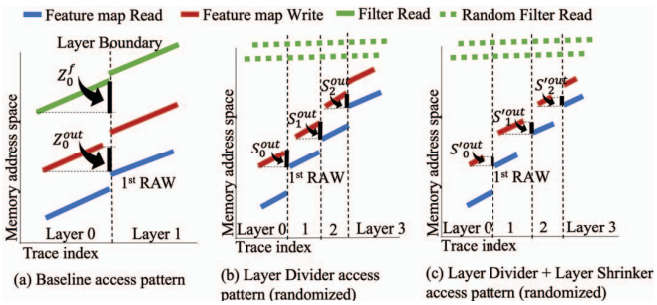


Fig. 8: DNNCloak access pattern obfuscation illustration.

a) Layer Divider Divide Rate Discussion: Figure 8 (b) depicts the access pattern after using the DNNCloak Layer Divider. The address space is randomized but we preserve the access pattern of feature maps before the randomization for illustrative purposes while the address randomization cannot hide the RAW dependency. In this case, the divide rate $R_d = 2$. We divide two DNN layers into four sublayers. Each sublayer needs to read the preceding sublayer's OFM, generating more layer boundaries exposed by RAW dependency, so that the attacker observes more layers. If we set a larger divide rate, it would be extremely difficult for the attacker to reverse the original structure even if the attacker knows the divide rate. As shown in table I, we use AlexNet as an example, when $R_d = 32$, there are 1.28×10^{13} possible DNN structures for reconstructing the observed layer sequence to the original sequence. Also, one layer sequence has dozens of possible dimension structures [10], making obtaining the original model structure extremely costly for attackers. On the other hand, with additional sublayers, the training cost increases significantly and only a few possible structures may achieve a similar accuracy as the original model, the accuracy of most structures decreases due to the overfitting issues. For example, the AlexNet achieves 57% top-1 validation accuracy with the original structure on CIFAR100 datasets; by dividing more sublayers, with the same training settings, the model accuracy decreases significantly, as shown in table I.

b) Layer Shrinker Buffer Size Discussion: In Figure 8 (b), the OFM size S_i^{out} is still exposed, and also we can still infer that $S_0^{out} + S_1^{out} = Z_0^{out}$ in this example. Figure 8 (c) enhances the defense with the Layer Shrinker. After buffering a part of OFM in LS-Buffer, the exposed ${S'_0}^{out}$ is less than S_0^{out} (${S'_0}^{out} + {S'_1}^{out} < Z_0^{out}$), so the attacker cannot obtain the real dimension of each layer preventing the reverse DNN attack from commencing. In our experiments, we set the LS-Buffer size to 16KB or 32KB and we expect the LS-Buffer size will be much smaller than the DNN accelerator's global buffer, which commonly ranges between 100KB and 500KB. Using VGG16 as an example, the size of the feature maps within a single layer during DNN inference is around 1MB to 6MB; however, because we implement the Layer Divider, the size of feature maps within each layer will be split by the R_d. When $R_d = 32$, the feature maps of each sublayer are about 32KB to 192KB, which means that the LS-Buffer can store a substantial portion of them and hide the actual dimension.

TABLE I: AlexNet with DNNCloak Protection.

Divide rate	Possible structures	Re-train accuracy
$R_d = 0$	24	57.30%
$R_d = 8$	5.53×10^8	43.56%
$R_d = 16$	8.94×10^{10}	11.56%
$R_d = 32$	1.28×10^{13}	0.91%

In sum, DNNCloak increases the exposed number of layer boundaries, limiting the attacker from getting all possible attack structures; moreover, the new structures result in lower accuracy; and DNNCloak hides the exposed dimension of each layer by buffering a portion of feature maps. Therefore, DNNCloak can mitigate the reverse DNN attacks effectively.

VI. Evaluation

A. Experimental Methodology

The evaluation takes three main steps: First, we use Keras [5], an open-source python library that provides a variety of DNN models, to run DNN different applications. Keras may be used to learn about the model structure, such as the layer sequence, layer type, and dimensions. As mentioned in section IV-C2, we use CIFAR10 as dataset when we evaluate the accuracy loss of the re-trained model.

Second, we implement DNN applications in C++, in which we can customize the layer structure and weight parameters. All operations of our program run on the local CPU. The main purpose of this step is to collect the access pattern of the DNN application. So, when running the C++ program, we record the addresses and access types of the operations related to the feature maps and filters. We have four models, VGG16, VGG19, AlexNet, and LeNet; we collect more than 2 billion memory accesses trace for each application.

Third, we conduct experiments on the USIMM simulator [1], and we evaluate the memory cycle time as performance results. USIMM is a cycle-accurate memory framework that reads an application trace as input and outputs a memory cycle. For the simulation, we use DDR4-1866 memory, which operates at a command rate of 933MHz, and a cycle time of 1.07ns. The memory configuration and timing is referred to [14]. Besides, we set the LS-Buffer latency as 10 cycles and the randomizer latency is 3 cycles [23]. The USIMM conducts trace-driven simulation so we utilize the access pattern trace generated in the second step as input for our tests.

B. Experimental Results

1) ***Results of LD and LS****:* The divide rate R_d and LS-Buffer size S_{buffer} have a direct effect on the results of Layer Divider and Layer Shrinker. We use four applications to examine the results with different R_d and S_{buffer}. Figure 9 illustrates the memory cycle results from the USIMM simulation. Each figure shows four different divide rate values, $R_d = 0, 8, 16, 32$, respectively. For each R_d, we use three different LS-Buffer size, $S_{buffer} = 0, 16, 32KB$. With the increased R_d, DNNCloak can provide higher obfuscation but has the expense of a long memory cycle. While increasing the S_{buffer} of the LS-Buffer can help reduce memory cycles. When $R_d = 32$ and the LS-Buffer size is 32KB, the memory cycle costs 1.28x, 1.14x, 1.06x and 1.11x more than the $R_d = 0$ and no LS-Buffer baseline in four applications.

2) ***Results of LOB****:* The Layer Obfuscator introduces extra dummy accesses in order to obfuscate the access intensities in different types of layers. As shown in figure 10 (a), we compare the memory cycle to the $R_d = 0$ and no LS-Buffer baseline, we choose $R_d = 32$ and $S_{buffer} = 32KB$ as our default Layer Divider (LD) and Layer Shrinker (LS) settings. LeNet only contains two Conv layers and two pooling layers, so the LOB introduces more extra accesses in percentage compared to other models. Our results show that, after implementing LOB with LD and LS, the total memory cycle increases 1.26x compare to the baseline.

3) ***Results of RPM****:* We use VGG16 as an example to show the RPM results with different pruning ratios in figure 10 (b). We modify the ratio of non-critical numbers by pruning the smallest value to zero. The left side of the Y-axis shows the memory cycle, and the right side shows the model accuracy loss after pruning. The model accuracy depicted by the line decreases slowly at the point of 0%, 20%, 40%, with only less than 0.1% accuracy loss; but when pruning 60%, the model accuracy loss increase to 4%, and after that, the accuracy will drop rapidly. When we prune more non-critical numbers, more memory access can be reduced with our RPM. As a result, at the point 40%, while preserving the model accuracy, VGG16 takes only 0.68x memory cycles compared to the case without pruning.

4) ***Overall DNNCloak performance****:* We demonstrate the DNNCloak schemes separately in the previous section. We show the trade-off between Divide Rate R_d and performance, the influence of LS-Buffer buffer size S_{buffer} and the trade-off between model accuracy and performance. In this section, we use $R_d = 32, S_{buffer} = 32KB$ with LOB and 0.1% accuracy loss with RPM 0.4 pruning ratio as our default DNNCloak setting and show the overall performance compared to the baseline. The baseline is running with no encryption and no obfuscation. We compare our proposed DNNCloak with full encryption, partial encryption (with 40% encryption ratio) and DNNCloak without RPM (pruning ratio =0) cases. The VGG16 and VGG19 use 3×3 matrix in filters, whereas AlexNet and LeNet mainly use 5×5 matrix filters, so the RPM performs better for VGG16 and VGG19 since one filter can be compressed into one data block. Moreover, the LOB decreases more for LeNet as explained in VI-B2. Therefore, as shown in figure 10 (c), our default DNNCloak reduces the memory cycle to 0.73x, 0.69x compared to the baseline for VGG16 and VGG19, while DNNCloak takes 1.02x, 1.08x more memory cycles than the baseline for AlexNet and LeNet.

On average, DNNCloak only takes 0.54x, 0.7x and 0.69x compared to full encryption, partial encryption and DNNCLoak without RPM, which are 1.59x, 1.22x and 1.26x compared to the baseline; and the memory cycle of our default DNNCloak is only 0.86x compared to the baseline.

VII. Conclusions

In this paper, we propose DNNCloak, a lightweight obfuscation scheme to mitigate reverse engineering attacks on DNN though access pattern side-channels. Our insight is that

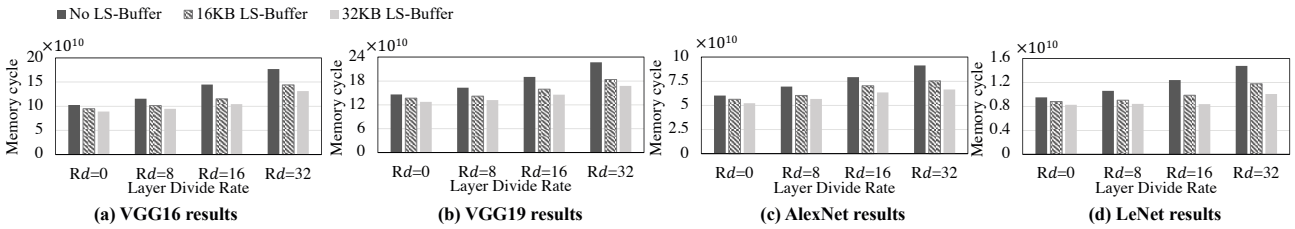


Fig. 9: Results of LD and LS in common DNN applications with different settings.

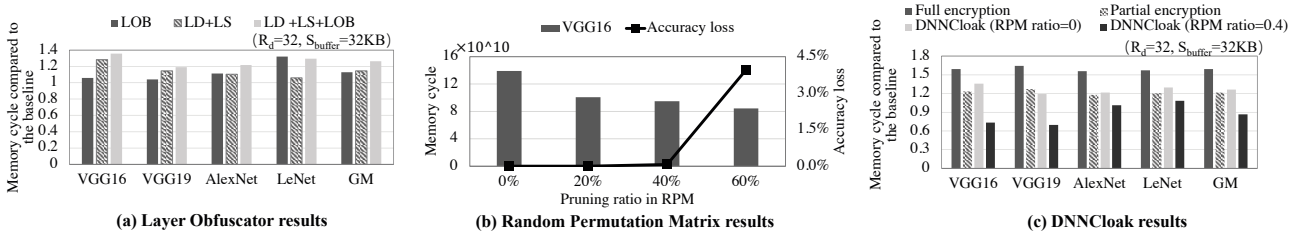


Fig. 10: Results of LOB, RPM and overall performance.

not all access patterns is necessary to be hidden and we identify the critical leakage hints that may leak the model structure during DNN reverse engineering attacks. Our design DNNCloak focus on hide those leakage hints only to achieve an efficient obfuscation purpose. DNNCloak has three methods that can obfuscate the original model structure, and also provides the random permutation matrix scheme to improve the performance. The experimental results of DNNCloak indicate that DNNCloak costs only 0.86x memory cycle time in average compared to the original model.

ACKNOWLEDGMENT

We thank the reviewers for their insightful comments. This research is supported in part by the National Science Foundation under grant CCF-2029014 and CNS-2152497.

REFERENCES

[1] N. Chatterjee et al. “Usimm: the utah simulated memory module”. In: *University of Utah, Tech. Rep* (2012).

[2] Y. Che et al. “Imbalance-aware scheduler for fast and secure ring oram data retrieval”. In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE. 2019, pp. 604–612.

[3] Y. Che et al. “Multi-range supported oblivious RAM for efficient block data retrieval”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020.

[4] Y.-H. Chen et al. “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks”. In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138.

[5] F. Chollet et al. *Keras*. https://keras.io. 2015.

[6] C. W. Fletchery et al. “Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs”. In: *HPCA*. 2014.

[7] O. Goldreich et al. “Software protection and simulation on oblivious RAMs”. In: *Journal of the ACM (JACM)* (1996).

[8] C. Gongye et al. “Reverse-engineering deep neural networks using floating-point timing side-channels”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020.

[9] X. Hu et al. “Deepsniffer: A dnn model extraction framework based on learning architectural hints”. In: *ASPLOS*. 2020.

[10] W. Hua et al. “Reverse engineering convolutional neural networks through side-channel information leaks”. In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018.

[11] T. Kohno et al. “CWC: A high-performance conventional authenticated encryption mode”. In: *International Workshop on Fast Software Encryption*. Springer. 2004, pp. 408–426.

[12] D. Lee et al. “An off-chip attack on hardware enclaves via the memory bus”. In: *29th {USENIX} Security Symposium*. 2020.

[13] J. Li et al. “NeurObfuscator: A Full-stack Obfuscation Tool to Mitigate Neural Architecture Stealing”. In: *arXiv preprint arXiv:2107.09789* (2021).

[14] S. Li et al. “DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator”. In: *IEEE Computer Architecture Letters* (2020).

[15] Y. Liu et al. “Delving into transferable adversarial examples and black-box attacks”. In: *arXiv:1611.02770* (2016).

[16] Y. Liu et al. “Mitigating reverse engineering attacks on deep neural networks”. In: *ISVLSI*. 2019.

[17] Y. Liu et al. “GANRED: GAN-based Reverse Engineering of DNNs via Cache Side-Channel”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 2020.

[18] Y. Long et al. “Understanding membership inferences on well-generalized learning models”. In: *arXiv preprint:1802.04889* (2018).

[19] A. J. Menezes et al. *Handbook of applied cryptography*. CRC press, 2018.

[20] S. J. Oh et al. “Towards reverse-engineering black-box neural networks”. In: *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer, 2019, pp. 121–144.

[21] A. Parashar et al. “Scnn: An accelerator for compressed-sparse convolutional neural networks”. In: *ACM SIGARCH Computer Architecture News* (2017).

[22] M. K. Qureshi. “Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping”. In: *MICRO*. 2018.

[23] M. K. Qureshi et al. “Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling”. In: *MICRO*. 2009.

[24] L. Ren et al. “Constants Count: Practical Improvements to Oblivious {RAM}”. In: {*USENIX*} *Security*. 2015.

[25] O. Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *IJCV* (2015).

[26] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* (2016).

[27] K. Simonyan et al. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[28] E. Stefanov et al. “Path ORAM: an extremely simple oblivious RAM protocol”. In: *CCS*. 2013.

[29] F. Tramèr et al. “Stealing machine learning models via prediction apis”. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 601–618.

[30] W. Xiong et al. “The Microsoft 2017 conversational speech recognition system”. In: *IEEE ICASSP*. 2018.

[31] M. Yan et al. “Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures”. In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.

[32] P. Zuo et al. “SEALing Neural Network Models in Encrypted Deep Learning Accelerators”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021.