Mitigating Reverse Engineering Attacks on Deep Neural Networks

Yuntao Liu, Dana Dachman-Soled, and Ankur Srivastava
University of Maryland, College Park
ytliu@umd.edu, danadach@ece.umd.edu, ankurs@umd.edu

***Abstract*—With the structure of deep neural networks (DNN) being of increasing commercial value, DNN reverse engineering attacks have become a great security concern. It has been shown that the memory access pattern of a processor running DNNs can be exploited to decipher their detailed structure [1]. In this work, we propose a defensive memory access mechanism which utilizes oblivious shuffle, address space layout randomization, and dummy memory accesses to counter such attacks. Experiments show that our defense exponentially increases the attack complexity with asymptotically lower memory access overhead compared to generic memory obfuscation techniques such as ORAM and is scalable to larger DNNs.**

I. Introduction

Recent years have seen an unprecedented development of artificial intelligence in which deep neural networks (DNN) have been a major driving force. Due to the difficulty of training high performance DNNs, the structure and weights of the DNN are a crucial intellectual property of modern machine learning based systems. The owner of the DNN model only wishes to provide I/O access to users without divulging the inner details. Unfortunately, even if the DNN structure is not explicitly given to the user, reverse-engineering attacks can be used to reconstruct it. In particular, it has been shown that the DNN structure can be easily reverse-engineered if the memory access pattern of the processor running the DNN is leaked [1]. This is a significant security concern. To the best of the authors' knowledge, no efficient countermeasure has been proposed. Although applying an oblivious RAM (ORAM) protocol is a well-established approach to hide the memory access pattern, it comes with very high memory access overheads [2]–[4]. Because running DNNs is a memory intensive task, the speed of the DNN running in hardware is mostly constrained by the number of memory accesses [5]–[7]. This makes ORAM-based memory access obfuscation impractical for DNNs.

Oblivious shuffle also provably obfuscates the address space with lower overhead than ORAM albeit with weaker theoretical guarantees [8] (detailed in Section III). In this work, we utilize oblivious shuffle to obfuscate a subset of the memory access patterns. The subset itself is customizable by the designer. A bigger subset (which could at most include the entire memory space) results in stronger obfuscation at the cost of higher memory access overheads. In addition, we use address space layout randomization (ASLR) on the entire memory space and add dummy memory access (DumMA) requests to the shuffled addresses for further improvements in security guarantees.

The contribution of this work is as follows:

- A novel defense strategy to obfuscate the processor's memory access pattern is proposed in order to reduce information leakage about the structure of the DNN being executed. This strategy utilizes three techniques: oblivious shuffle, address space layout randomization (ASLR), and dummy memory access (DumMA). Although these techniques have been existing, our innovation lies in combining them strategically to thwart the attack with low overhead.
- A modified attack based on that in [1] is formulated to reverse-engineer the DNN structure in the presence of our defense in order to evaluate the security of our defense.
- Experimental results show that the complexity of the modified attack is very high thereby demonstrating the effectiveness of our defense.
- It is also shown that the memory access overhead of our defense is very low and does not increase with the DNN depth, making our approach scalable to deeper models.

II. DNN Reverse-Engineering

The recent work of Hua, Zhang, and Suh [1] illustrates an elegant optimization theoretic attack based on the *memory access side-channels* of systems running DNNs. The attack model considered is as follows. The owner of the DNN model wants to enable the user to run the model on her own processor (*e.g.* a CPU, GPU, or DNN accelerator) without exposing the structure of the DNN model. The attacker is considered to be the user who is **honest-but-curious**: she wants to know the details of the model but does not interfere with the normal execution of the DNN model. The processor is considered as secure, *i. e.* the attacker cannot observe or interfere with the processor's internal operations.

However, the attacker is able to observe the **memory access patterns** of the processor, *i. e.* a transcript of its memory accesses including the accessed addresses, the access types (*i. e.* read or write), and the time of each access. The attacker also knows the input and output of the DNN (since she has I/O access to the DNN). As shown in [1], a reverse-engineering attack can be formulated under this attack model and the architecture of a DNN can be extracted.

A. Attack Setup

The specific type of deep neural networks (DNN) of interest to us is convolution neural networks. They have found significant applications in several machine learning problems dealing with classification. These networks comprise two types of layers: **convolutional (Conv)** layers and **fully connected (FC)** layers. [1] Each layer transforms a set of input neurons, called the **input feature map (IFM)**, into a set of output neurons, called the **output feature map (OFM)**. The OFM of the previous layer is the IFM of the next layer. Figure 1 illustrates the structure of a Conv layer. The structure of a Conv layer can be described by a set of hyper-parameters which are listed in Table I. The structure of a fully connected layer is much simpler. If layer i is an FC layer, its IFM and OFM are vectors of length z_i^{in} and z_i^{out}, respectively. A 2-D weight matrix of dimension $z_i^f = z_i^{in} \times z_i^{out}$ transforms the IFM to the OFM.

[1] Pooling layers are considered as a part of the Conv layers as opposed to separate layers since the processor writes the OFM to the memory after pooling (if the pooling layer exists) and the feature map sizes before pooling are not observable to the attacker. Activation functions are not considered in this work since they only modify the value of neurons not the structure of the DNN.

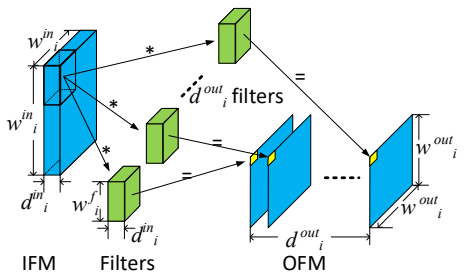

Fig. 1. Illustration of a Conv layer. "*" indicates the inner product, each of which computes an output neuron.

Parameter	Definition
w_i^{in}, w_i^{out}	width of the IFM/OFM of layer i
d_i^{in}, d_i^{out}	depth of IFM/OFM of layer i
z_i^{in}, z_i^{out}, z_i^f	size of IFM/OFM/filter of layer i
P_i	indicator of whether pooling exists in layer i
f_i^{conv}, f_i^{pool}	filter width of convolution/pooling (if existing) of layer i
s_i^{conv}, s_i^{pool}	stride of convolution/pooling (if existing) of layer i
p_i^{conv}, p_i^{pool}	padding of convolution/pooling (if existing) of layer i

TABLE I
LIST OF HYPER-PARAMETERS OF EACH LAYER

As in the attack model of [1], the DNN model is stored in a virtual address space that starts from 1 and each neuron or weight takes exactly 1 address to store. The way that the neurons and weights are aligned in the memory is as follows: The first feature map (*i. e.* the IFM of layer 0, of size z_0^{in}) starts from address 1 and ends at z_0^{in}. The second feature map (the OFM of layer 0 and the IFM of layer 1, of size z_0^{out} which equals z_1^{in}) starts at $z_0^{in}+1$, and so on. The weights are stored in a separate area of memory and organized in a layer-by-layer configuration (similar to the neurons).

B. Attack Methodology

The attack to reverse-engineer the structure of a DNN consists of 3 phases: determining the layer boundaries in the memory traces, solving for the feasible DNN structures which fit the memory trace, and training each feasible structure for the best match.

1) Phase 1: In this phase, the attacker determines the layer boundaries in the memory access transcript leaked by side-channels and obtains the IFM, OFM, and filter sizes of each layer. This process is illustrated in Figure 2.

The **layer boundaries** are expressed in terms of the clock cycles at which the first memory access of each layer occurs. Let $c_i,\ i \in [L]$ be the first clock cycle of layer i where L is the number of layers. The attacker determines the boundaries between layers by observing the first occurrence of **read-after-write (RAW)** in the memory. This

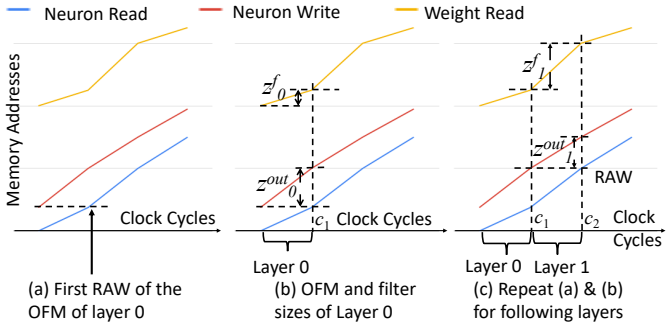

Fig. 2. Illustration of the attack in [1] per Section II-B.

Benchmark	input		# layers			attack complexity metrics	
	w_0^{in}	d_0^{in}	Conv	FC	Total	# IFPs solved	# feasible structures
DNN 1	28	1	2	2	4	4	4
DNN 2	32	1	3	2	5	5	6
DNN 3	32	3	4	3	7	8	1
DNN 4	64	3	5	3	8	9	8

TABLE II
BENCHMARK DNNS AND ATTACK COMPLEXITY USING THE ATTACK IN [1]

is illustrated in Figure 2(a). Layer 0 writes its OFM to the memory and layer 1 needs to read the same memory location to get it as its IFM. Therefore, the first occurrence of RAW indicates that layer 0 has finished and layer 1 has started, thereby leaking c_1 (Figure 2(b)). In this way, by counting how many addresses have been written to before c_1, the attacker observes the OFM size of layer 0 z_0^{out} (Figure 2(b)). Similarly, the attacker can also observe the filter size of layer 0 z_0^f. The first RAW of layer 1's OFM marks the beginning of layer 2 (Figure 2(c)). By repeating the above procedure for all the subsequent layers, the IFM sizes, OFM sizes, filter sizes, the starting clock cycles of all the following layers can be observed.

2) Phase 2: After obtaining feature map and filter sizes in the previous phase, the attacker tries to obtain *all the feasible structures* of the DNN that conform with the observed access pattern. Each structure is described by a combination of the hyper-parameters of every layer (as listed in Table I). These hyper-parameters are obtained by solving an **integer feasibility program (IFP)** problem that captures the relationship among the hyper-parameters for each layer with the information obtained from phase 1. The IFP is defined by the following equations:

$$z_i^{in} = (w_i^{in})^2 \times d_i^{in}, \quad z_i^{out} = (w_i^{out})^2 \times d_i^{out} \tag{1}$$

$$z^f = (f_i^{conv})^2 \times d_i^{in} \times d_i^{out} \tag{2}$$

$$w_i^{out} = \frac{\frac{w_i^{in} - f_i^{conv} + p_i^{conv}}{s_i^{conv}} + 1 + P(p_i^{pool} - f_i^{pool})}{s_i^{pool} \times P + \bar{P}} \tag{3}$$

$$s_i^{conv} \le f_i^{conv} \le \frac{w_i^{in}}{2} \tag{4}$$

$$s_i^{pool} \le f_i^{pool} \le \frac{w_i^{in} - f_i^{conv} + p_i^{conv}}{s_i^{conv}} + 1 \tag{5}$$

$$p_i^{conv} < f_i^{conv}, \quad p_i^{pool} < f_i^{pool} \tag{6}$$

3) Phase 3: After all the possible structures are obtained, the attacker trains each structure and picks the one with the highest accuracy as the final outcome. It was shown in [1] that the feasible structures obtained from the memory traffic were very few thereby significantly reducing the training effort.

C. Attack Complexity and Practicality

In this work, we consider 4 DNN benchmarks which are listed in Table II. The complexity of the attack is measured using two metrics: the number of **IFP problems solved** and the total number of **feasible DNN structures**. The former represents the hardness involved with obtaining the set of feasible DNN structures (essentially the complexity of Phase 2). The latter represents the amount of training effort needed by the attacker to pick the best model (essentially Phase 3).

We wrote a simulator to generate a processor's memory trace. The processor's memory trace is reverse-engineered using the above-described attack method. The complexity of the attacks on the benchmark DNNs is also shown in Table II. As seen, both metrics are low for all the benchmarks, indicating the low complexity of the attack.

It is important to note that the "exact" neural network with exactly the same weights and topology may not be the one synthesized by

this attack. However, the attacker's objective would still be achieved since she would still be able to get substantially accurate classification performed by synthesizing the model based on the one running on the processor.

III. Cryptographic Preliminaries

The effectiveness of the above-mentioned attack necessitates a defense mechanism that reduces the information leakage of the DNN structure in the memory access patterns. Hiding memory access patterns is a well-studied problem and has been formalized via the notion of **Oblivious RAM (ORAM)** schemes. An ORAM scheme can be used to obfuscate the memory access patterns of any input RAM program and provides the strong theoretical guarantee that the obfuscated memory access patterns reveal no information about the input program [9]. However, even the state-of-the-art ORAM protocol [2]–[4] incurs an **access overhead** of $\Omega(\log N)$, *i. e.* the average number of memory accesses that have to be performed in order to access a single address in the original program is at least $\log N$, where N is the total number of address.

In our work, instead of ORAMs, we consider a different approach called **oblivious shuffle** whose overhead is much lower [10]. We define oblivious shuffle below followed by an explanatory example.

Definition 3.1 (Oblivious Shuffle): A shuffle algorithm is an algorithm of the form $(\mathsf{Enc}(\pi(A)), \alpha) \leftarrow \mathsf{Shuffle}(A, \mathsf{Enc}, \pi)$ where A is an input array, Enc is a secure encryption algorithm, and π is a random, predetermined permutation function. The output of $\mathsf{Shuffle}$ is an encryption of the permutation of A according to π and a memory access transcript α. The $\mathsf{Shuffle}$ algorithm is an oblivious shuffle if α is independent of π.

π gets obfuscated by an oblivious shuffle. An example of oblivious shuffle algorithm is as follows. Let A be an array of length N, π be a permutation function over $[N]$, π^{-1} be the inverse function of π, and Enc be a secure encryption algorithm. We assume for simplicity that the index of an element in A is equal to its address in the memory. A simple oblivious shuffle algorithm is as follows:

for i in $[N]$ **do**
  Read address i for $A[i]$ and store $A[i]$ in the secure on-chip memory of the processor
end for
for i in $[N]$ **do**
  Write $\mathsf{Enc}(A[\pi^{-1}(i)])$ to address i
end for

Using the above algorithm, *the attacker will always see the same memory access pattern regardless of* π, which, in this case, is a read sequence followed by a write sequence, both in the address order of $0, 1, \ldots, N-1$. Hence the attacker cannot decipher π. Reference [8] proposed the state-of-the-art oblivious shuffle algorithm, called the *Melbourne shuffle*, which is efficient and scalable: it only requires $O(\sqrt{N})$ private memory (*i. e.* the memory not observable to the attacker) to shuffle an array of size $O(N)$ as proven in Theorem 5.1 in [8].

In this work, the permutation function π we choose 'looks' random and utilizes the internal randomness of the processor (which is fixed for the same shuffle but can vary for different shuffles). Similar to the above example, the attacker will also see a fixed memory access pattern α of Melbourne shuffle no matter what π is.

Under this condition, if a processor is running a DNN model and switches between Melbourne shuffle phases and regular DNN phases, there will be three types of phases in the memory access pattern: (i) the Melbourne shuffle, (ii) the DNN accesses *inside* the shuffled addresses, and (iii) the DNN access *outside* the shuffled addresses. The attacker can distinguish these phases because, no matter what memory access pattern is generated by the DNN application, that of the Melbourne shuffle will always be α. The attacker can hence also observe the addresses that are shuffled. Note, however, that the attacker has no information about π, or the actual memory addresses accessed by the DNN application during type (ii) phases since she does not know the internal randomness of the processor, nor does α leak any information of π.

Running DNNs is a memory-intensive task where every neuron and weight needs to be accessed. To compare the memory access overhead of ORAM and that of the Melbourne shuffle, we take an array A of length N and require that every element in A be accessed once. With ORAM, the total # accesses will be $\Omega(N \log N)$. With Melbourne shuffle, the memory accesses consists of two parts: those of the shuffle and those of the actual accesses. The Melbourne shuffle takes $O(N)$ memory accesses. Unlike the ORAM, once the oblivious shuffle is completed, there is no additional access overhead: one simply needs to access the new address. Hence the total # accesses will be $O(N) + N = O(N)$. *The overhead of the Melbourne shuffle is therefore a constant multiplicative factor*, making it asymptotically lower than that of the ORAM.

IV. Defense Methodology

In this section, we propose a memory access strategy for processors to run DNN models with minimal leakage of structural information. The defense should fulfill two competing objectives:

- The resulting attack complexity should be very high.
- The memory access overhead should be low.

As discussed in Sec. III, using ORAM will satisfy the first objective but fail the second one. In order to achieve both objectives, our proposed defense strategy utilizes 1) the *Melbourne shuffle* , 2) *address space layout randomization (ASLR)* [11], and 3) adding *dummy memory accesses (DumMA)*. A modified attack based on [1] to find the feasible structures of the DNN is also formulated in order to evaluate the security of our defense.

A. Utilizing Oblivious Shuffle

In order to obfuscate all the layer boundaries, a DNN with L layers needs $L-1$ oblivious shuffles (one for each layer boundary) during its execution. Note that we do not need to shuffle the entire memory: in the i-th shuffle, only the memory addresses accessed near c_i need to be shuffled (recall that c_i is the clock cycle at the beginning of layer i). Varying the number of shuffled addresses enables us to explore a spectrum of trade-offs between the memory access overhead and the security of the defense. In this subsection, we present how to determine when and where to shuffle and model the attacker's knowledge based on the new memory access pattern.

1) Oblivious Shuffle Strategy: We use the following method to determine where and when to shuffle. For the reasons described in Section III, we assume a strong attacker who can distinguish whether an access is a regular DNN-based request vs. a Melbourne shuffle request. Note that this assumption only strengthens the attacker and therefore designs in this threat model yield *more secure* strategies. In our explanation below, we express the timescale in terms of the clock cycles in the *original memory access pattern* (such as in Figure 2) and ignore the clock cycles that are spent on Melbourne shuffle as if it is done instantly. *In the rest of this paper, all the mentions of "memory accesses" are referred to those of the DNN model, NOT those of Melbourne shuffle.*

Step 1: determining the shuffle budget. In order to control the memory access overhead of the Melbourne shuffle, we shuffle at most 2^{b_s} addresses in each shuffle. b_s is called the **shuffle budget**.

Step 2: when to shuffle. In order to obfuscate c_i, *all the memory addresses that are accessed within a certain range of clock cycles*

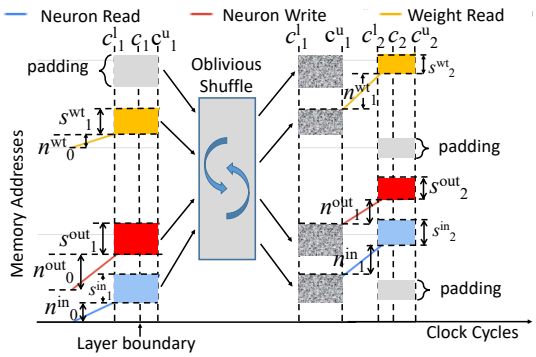

Fig. 3. Illustration of oblivious shuffle in the memory access pattern of a DNN. The variables that are observable to the attacker are also illustrated.

containing c_i are obfuscated using Melbourne shuffle. To this end, we determine the above-mentioned clock cycle range with an upper bound and a lower bound of c_i, denoted as c_i^l and c_i^u, respectively. We take $\hat{c}_i^l \sim U(c_i - 2^{b_s}, c_i)$ and $c_i^l = \lceil \hat{c}_i^l \rceil$ where U stands for the uniform distribution (assuming there is at most one memory access per clock cycle). Let $c_i^u = \lfloor c_i^l + 2^{b_s} \rfloor$. This makes sure that the total # memory address accessed from c_i^l to c_i^u does not exceed 2^{b_s}.

All the memory addresses that are accessed within $[c_i^l, c_i^u]$ are to be shuffled. We illustrate this in Figure 3 to which we encourage the readers to refer as the explanation proceeds. We call the set of shuffled memory addresses at c_i^l the **shuffled regime** i, which include s_i^{in} input neurons addresses, s_i^{out} output neurons addresses, s_i^{wt} weight addresses, and a set of additional addresses.These additional addresses are randomly chosen and can be any address in the memory space allocated for the DNN, as shown in Figure 3. This padding action will also enable us to add dummy accesses in the shuffled regimes to improve security as will be shown later.

Step 3: choosing the permutation π. Although the Melbourne shuffle will not leak information about π, π should 'look' random enough in order to obfuscate the memory accesses within the shuffled regimes. In addition, π must be easy to compute, otherwise we would spend too much time computing π. To meet these requirements, we use the *Feistel-based format-preserving encryption* [12] algorithm for π.

2) Information Leakage: We model the attacker's *best-case* knowledge in order to analyze the *worst-case* security guarantees. As noted before, we assume a strong attacker who can tell the difference between a Melbourne shuffle access and a regular DNN access. Therefore, c_i^l and the addresses within the shuffled regime i are known to the attacker. She can also infer c_i^u since after c_i^u, the memory accesses of the DNN will come out of the shuffled regime. As described in Section II, by default, the neurons are stored in consecutive addresses and the weights too. In line with the *best-for-the-attacker* principle, we assume that each kind of memory access (*i. e.* neuron read, neuron write, or weight read) within each shuffled regime is of consecutive addresses. In this case, by calculating the difference of accessed addresses before c_i^l and after c_i^u, the attacker is able to infer s_i^{in}, s_i^{out}, and s_i^{wt}.

The memory access pattern outside the shuffled regimes are directly visible to the attacker. Let n_i^{in}, n_i^{wt}, and n_i^{out} denote the # read neurons, # read weights, and # written neurons, respectively, between clock cycle $c_i^u + 1$ and $c_{i+1}^l - 1$ (essentially the region between two adjacent shuffled regimes, note that in this region layer i is active) as illustrated in Figure 3. Let $\hat{s}_i^{in}$ and $\hat{s}_i^{wt}$ be the # shuffled input neurons and weights in the shuffled regime i, respectively, that are accessed *before* c_i (*i. e.* belong to layer $i-1$). In order to find all feasible structures of layer i, the attacker needs to enumerate all the possible (integer) combinations of $c_i \in [c_i^l, c_i^u]$ and $c_{i+1} \in [c_{i+1}^l, c_{i+1}^u]$ (since she does not know exactly where the c_i, c_{i+1} lie within these boundaries). Let $r_{t_1}^{t_2}$ and $w_{t_1}^{t_2}$ be the # read and written addresses, respectively, within $[t_1, t_2]$. The following equations hold:

$$r_{c_i}^{c_{i+1}} = z_i^{in} + z_i^f \tag{7}$$

$$w_{c_i}^{c_{i+1}} = z_i^{out} \tag{8}$$

$$\hat{s}_{i+1}^{in} = z_i^{in} - n_i^{in} + (s_i^{in} - \hat{s}_i^{in}) \geq 0 \tag{9}$$

$$\hat{s}_{i+1}^{wt} = z_i^{f} - n_i^{wt} + (s_i^{wt} - \hat{s}_i^{wt}) \geq 0 \tag{10}$$

Equation (7) states that the total # read addresses between the layer boundaries is equal to the summation of the IFM size and the filter size. Similarly, (8) is based on the fact that the OFM is the only thing that is written back to the memory. Equations (9) and (10) are because the read accesses of each layer consist of 3 parts: (a) those in the previous shuffled regime (i), (b) those that are not shuffled, and (c) those in the current shuffled regime ($i+1$), and those in (c) must be non-negative.

Equations (7) through (10) gives the possible combinations of z_i^{out} and z_i^f. Each possible combination is plugged in the IFP problem in Equations (1) through (6). The more possible combinations of z_i^{out} and z_i^f, the more IFPs to be solved and hence the greater attack complexity. In order to increase the # possible combinations and hence attack complexity, we propose to use ASLR and DumMA to relax the constraints on z_i^{out} and z_i^f imposed by Equations (7) through (10).

B. Address Space Layout Randomization

ASLR was initially proposed to counter the buffer overflow attack [11]. In our work, ASLR is only done once at compile time to randomize the entire memory space. Each address is permuted using a permutation function π_{init} which maps an address addr to be initially stored in address $\pi_{init}(\text{addr})$. We use the same type of algorithm for π_{init} as the π for the Melbourne shuffle. In this way, the access pattern even outside the shuffled regimes will look random and the continuity of the address space is broken. Therefore, the attacker is not able to infer n_i^{in}, n_i^{out}, or n_i^{wt}. As a result, Equations (9) and (10), which require these variables, are not applicable any more. In this way, the constraints on z_i^{out} and z_i^f are reduced to only Equations (7) and (8). This will result in more possible combinations of z_i^{out} and z_i^f and hence force the attack to solve more IFPs. Note that ASLR does not increase the number of memory accesses at run time since it works only as a mapping from the requested address to the actual address. Also note that ASLR just by itself does not mitigate the RAW type attack and needs to be combined with oblivious shuffle.

C. Dummy Memory Accesses

In this technique, we add dummy memory access within the shuffled regimes. When dummy memory accesses (DumMA) exist in the shuffled regimes, the attacker cannot tell a real access from a dummy one. However, she still gets an *upper bound* of # real read/write addresses since they must not exceed the total # read/write addresses (*i. e.* real+dummy). One question is how many dummy accesses should be added. This is answered as follows. Since repeated accesses to *the same address of the same type* are observable, these accesses will not increase the upper bound of # real read/write addresses and do not improve the level of obfuscation. Therefore, there is no need to add more dummy accesses when every address in the shuffled regime is both read and written once.

Techniques	Availability to the attacker		Equations
	$s_i^{in}, s_i^{out}, s_i^{wt}, n_i^{in}, n_i^{out}, n_i^{wt}$	$r_{c_{i-1}}^{c_i}, w_{c_{i-1}}^{c_i}$	
OS	Yes	Exact	(7) $\sim$ (10)
OS + ASLR	No	Exact	(7), (8)
OS + DumMA	Yes	Upper bound	(9) $\sim$ (12)
All the above	No	Upper bound	(11), (12)

TABLE III
THE INFORMATION LEAKED TO THE ATTACKER UNDER VARIOUS COMBINATIONS OF DEFENSE TECHNIQUES

1) DumMA Without ASLR: In this case, the attacker is able to infer each type of shuffled addresses: s_i^{in}, s_i^{out}, and s_i^{wt} because they only rely on c_i^l, c_i^u. For this reason, Equations (9) and (10) still hold. However, Equations (7) and (8) need to be changed to reflect the "upper bound" effect caused by DumMA: the # of reads and writes between the layer boundaries are the upper bounds of $z_i^{in} + z_i^f$ and z_i^{out}, respectively (since many of these accesses are dummies).

$$r_{c_i}^{c_{i+1}} \geq z_i^{in} + z_i^f \tag{11}$$

$$w_{c_i}^{c_{i+1}} \geq z_i^{out} \tag{12}$$

Compared to Equations (7) and (8), (11) and (12) change equalities to inequalities (with the equal sign), and thus increasing the possible combinations of z_i^{out} and z_i^f.

2) DumMA With ASLR: Due to ASLR, Equations (9) or (10) does not hold any more for the same reason as described in Sec. IV-B. z_i^{out} and z_i^f are hence only constrained by Equations (11) and (12).

D. Summary of Defense Techniques

Three techniques have been introduced in the formulation of our defense: oblivious shuffle (OS), address space layout randomization (ASLR), and dummy memory accesses (DumMA). Each OS obfuscates the accessed memory addresses within a certain range of clock cycles containing a layer boundary. The following information remains leaked to the attacker: (i) the nature of each memory access outside the shuffled regimes, (ii) the # shuffled input neurons s_i^{in}, output neurons s_i^{out}, and weights s_i^{wt} in each shuffled regime, and (iii) the (actual) # read and written addresses of the DNN model. (i) and (ii) are obfuscated by ASLR and (iii) by DumMA. The information leakage under four cases is summarized in Table III: OS only, OS + ASLR, OS + DumMA, and OS + ASLR + DumMA.

E. Attacking the Proposed Defense

As mentioned earlier, we assume that the attacker knows which defense techniques are in place and is able to attack accordingly. The new attack of layer i is shown using the following pseudo-code:

for $(c_i, c_{i+1}) \in [c_i^l, c_i^u] \times [c_{i+1}^l, c_{i+1}^u]$ **do**
  for Each feasible structure of layer $i-1$ ending at $c_i - 1$ **do**
    Find all the possible combinations of z_o and z_f according to the equations summarized in Table III.
    for Each possible (z_o, z_f) pair **do**
      Solve the IFP defined by Equations (1) through (6).
  Concatenate all the found feasible structures of layer i to the currently used structure of layer $i-1$.

When the above algorithm finishes for the last layer, all the feasible structures of the DNN will be obtained.

V. EXPERIMENTS AND RESULTS

In this section, we evaluate the effectiveness of our proposed defense strategy using the complexity of the *modified* attack and measure the memory access overhead. The attack complexity is evaluated in the same way as described in Section II-C, with the two metrics being the number of IFP problems (defined by Equations (1) $\sim$ (6)) (Phase 2) to be solved and the total number of feasible DNN structures that need to be trained and evaluated (Phase 3).

Benchmark		$b_s = 6$	$b_s = 7$	$b_s = 8$	$b_s = 9$
DNN 1	Without DumMA	4.10%	10.89%	10.94%	33.63%
	With DumMA	4.36%	11.35%	12.07%	35.21%
DNN 2	Without DumMA	8.85%	25.20%	36.24%	86.66%
	With DumMA	9.31%	26.44%	37.95%	87.58%
DNN 3	Without DumMA	2.38%	3.72%	9.03%	16.42%
	With DumMA	2.49%	3.99%	9.41%	17.16%
DNN 4	Without DumMA	1.12%	1.83%	5.18%	13.97%
	With DumMA	1.76%	3.12%	6.79%	17.41%
Average	Without DumMA	4.11%	10.41%	15.35%	37.67%
	With DumMA	4.48%	11.23%	16.56%	39.34%

TABLE IV
THE OVERHEAD OF OUR PROPOSED DEFENSE

The shuffle budget b_s ranges from 6 to 9 in our experiments. Under each b_s, we generate the memory traces for each combination of defense techniques. When DumMA is used, we add dummy accesses into each shuffled regime such that each shuffled address is both read once and written once.

The two complexity metrics of the 4 benchmarks are reported in Figure 4. We observe that the combination of all three techniques yields the highest security level:

1) Both metrics are many orders of magnitude better than any other combination of techniques within the same benchmark and under the same shuffle budget.
2) For each benchmark, both metrics grow exponentially with the shuffle budget.
3) The # possible structures tend to grow exponentially as the DNN gets deeper.

The overheads of our proposed defense under each shuffle budget from 6 to 9 are listed in Table IV. As seen, very high attack complexity can be achieved at the cost of low access overheads. Moreover, the memory access overhead does not increase when the DNN gets larger, making our technique easily scalable to larger DNN models. This is because the overhead is roughly determined by the ratio of the size of each shuffled regime to that of each layer. This scalability is a key advantage of our approach over ORAM. ORAM requires an $\Omega(\log N)(\times 100\%)$ access overhead where N is the size of the memory (and must be least the size of the DNN), which means that the access overhead must increase as the DNN becomes larger. The effectiveness and scalability of our defense strategy make it practical to defend the reverse engineering attacks on DNNs. Note that the secure encryption algorithms (for which we use AES) and the permutation function π (which is a simple transformation from AES) are not considered as significant sources of overheads because AES accelerators have been integrated into most processor architectures nowadays which allow very efficient computation of AES functions.

VI. RELATED WORK

Research on DNN model stealing attacks has become popular in the recent two years. Tramèr *et al.* proposed to extract DNN model hosted in servers through prediction APIs [13]. Juuti *et al.* proposed a statistical technique to detect model extraction [14]. Cache side-channels (using co-located processes with the DNN) [15], [16], power and electromagnetic side-channels [17], [18], and timing side-channel [19] have also been exploited to extract the DNN model or the input data. Our attack model is based on the memory access pattern of normal DNN inference which does not require any additional query. Hence detection approaches such as the one in [14] do not apply to this attack model. Many papers have suggested using homomorphic encryption (HE) to compute neural networks [20]–[23]. However, HE does not hide memory access patterns and is not a countermeasure against this attack either. Others have suggested secure multi-party computation (MPC) to protect both the data and the neural models [24]–[26]. Unfortunately, MPC needs to distribute the computation

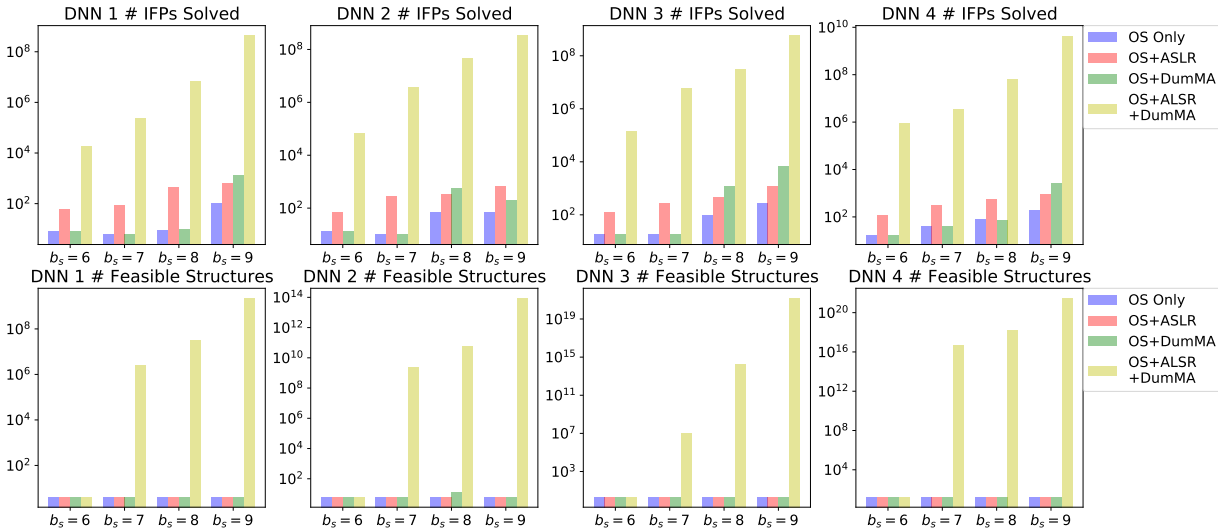

Fig. 4. # IFPs solved and # feasible structures of the DNN benchmarks under various defense techniques

to multiple computers and is not applicable to our case where the DNN is run on a local processor.

VII. Conclusion

A novel defense strategy against the reverse engineering on DNNs is proposed in this paper. The targeted attack model analyzes the memory access pattern of the processor running the DNN and solves an integer feasibility program to obtain all the possible structures of each layer. In our defense strategy, three techniques are utilized to obfuscate the memory access pattern, including oblivious shuffle, address space layout randomization, and dummy memory access. A modified attack based on the original attack is also formulated in order to evaluate the security of the proposed defense. Experiments show that, by combining all the three defense techniques, very high attack complexity can be achieved with low overheads. It is also shown that our defense approach easily scales to larger DNN models. Therefore, we conclude that the DNN reverse engineering attacks based on memory access patterns can be effectively countered using our proposed defense approach.

Acknowledgments

This work is supported by AFOSR MURI under Grant FA9550-14-1-0351 and Northrop Grumman Corporation and University of Maryland Seedling Grant.

References

[1] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 4.

[2] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," *Journal of the ACM (JACM)*, vol. 65, no. 4, p. 18, 2018.

[3] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "Panorama: Oblivious RAM with logarithmic overhead," *IACR Cryptology ePrint Archive*, vol. 2018, p. 373, 2018, to appear in FOCS 2018. [Online]. Available: https://eprint.iacr.org/2018/373

[4] G. Asharov, I. Komargodski, W. Lin, K. Nayak, and E. Shi, "Optorama: Optimal oblivious RAM," *IACR Cryptology ePrint Archive*, vol. 2018, p. 892, 2018. [Online]. Available: https://eprint.iacr.org/2018/892

[5] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[6] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[7] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 548–560.

[8] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The melbourne shuffle: Improving oblivious storage in the cloud," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2014, pp. 556–567.

[9] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Annual Cryptology Conference*. Springer, 2010, pp. 502–519.

[10] M. T. Goodrich and M. Mitzenmacher, "Anonymous card shuffling and its applications to parallel mixnets," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2012, pp. 549–560.

[11] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in *USENIX Security Symposium*, 2012, pp. 475–490.

[12] M. Bellare, P. Rogaway, and T. Spies, "The ffx mode of operation for format-preserving encryption," *NIST submission*, vol. 20, 2010.

[13] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 601–618.

[14] M. Juuti, S. Szyller, A. Dmitrenko, S. Marchal, and N. Asokan, "Prada: Protecting against dnn model stealing attacks," *arXiv preprint arXiv:1805.02628*, 2018.

[15] M. Yan, C. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn dnn architectures," *arXiv preprint arXiv:1808.04761*, 2018.

[16] S. Hong, M. Davinroy, Y. Kaya, S. N. Locke, I. Rackow, K. Kulda, D. Dachman-Soled, and T. Dumitraş, "Security analysis of deep neural networks operating in the presence of cache side-channel attacks," *arXiv preprint arXiv:1810.03487*, 2018.

[17] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi neural network: Using side-channels to recover your artificial neural network information," *arXiv preprint arXiv:1810.09076*, 2018.

[18] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. Xu, "I know what you see: Power side-channel attack on convolutional neural network accelerators," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 393–406.

[19] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing neural networks via timing side channels," *arXiv preprint arXiv:1812.11720*, 2018.

[20] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Annual International Cryptology Conference*. Springer, 2018, pp. 483–512.

[21] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network." *IACR Cryptology ePrint Archive*, vol. 2017, p. 35, 2017.

[22] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning*, 2016, pp. 201–210.

[23] P. Xie, M. Bilenko, T. Finley, R. Gilad-Bachrach, K. Lauter, and M. Naehrig, "Crypto-nets: Neural networks over encrypted data," *arXiv preprint arXiv:1412.6181*, 2014.

[24] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 2.

[25] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 19–38.

[26] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 619–636.