DNN Architecture Attacks via Network and Power Side Channels

Yuanjun Dai, Qingzhe Guo, and An Wang

Case Western Reserve University
{yxd429, qxg88, axw474}@case.edu

Abstract. The increasing complexity of machine learning models drives the emergence of Machine-Learning-as-a-Service (MLaaS) solutions provided by cloud service providers. With MLaaS, customers can leverage existing data center infrastructures for model training and inference. To improve training efficiency, modern machine learning platforms introduce communication optimization mechanisms, which can lead to information leakage. In this work, we present a network side channel based attack to steal model sensitive information. Specifically, we leverage the unique communication patterns during training to learn the model architectures. To further improve accuracy, we also collect information from software based power side channels and correlate it with the information extracted from network. Such temporal and spatial correlation helps reduce the search space of the target model architecture significantly. Through evaluations, we show that we can achieve more than 90% accuracy for model hyper-parameters reconstruction. We also demonstrate that our proposed attack is robust against background noise by evaluating with memory and traffic intensive co-located applications.

1 Introduction

The demand for artificial intelligence and machine learning has grown significantly over the past decade. This growth has been fueled by advances in machine learning technologies and the ability to leverage hardware acceleration. More complex models, such as deep neural networks (DNN), are required to increase the accuracy of machine learning-based solutions so that they become feasible for more complex applications. As the demand for complex model training and execution outpaces the growth of computation power provided by a single machine, it has become essential to leverage distributed computing infrastructures. Driven by this demand, Machine-Learning-as-a-Service (MLaaS), a collection of various cloud-based platforms that use machine learning tools to provide machine learning solutions, is emerging. Through MLaaS, clients can submit their models and training datasets to service providers (e.g., an Amazon and Google cloud service) and leverage their infrastructures for model training and inference.

Many researchers have realized that data security and privacy are significant concerns for these MLaaS systems, given that both model parameters and datasets are valuable assets due to their critical roles. A large body of work has been dedicated to stealing sensitive model information or data during machine learning inference. Such attacks include member inference attacks [1, 2] and attacks for stealing sensitive model information [3, 4]. But little has been done from the MLaaS platform perspectives – whether a new attacking surface specific to MLaaS systems exists and can be leveraged to launch attacks. Many MLaaS services are built on the fundamental building blocks of the existing cloud infrastructures. For example, Amazon SageMaker relies on containers and EC2 instances to train ML models. Such instances are also leveraged by AWS to host other services, thus providing opportunities for attackers to steal sensitive model information. Our work is among the first ones to explore answers to this question.

In this work, we identify and explore MLaaS system-specific attacking surfaces to steal sensitive information while training DNN models, mainly via network side channels. Our key observation is that training models in a distributed manner generally requires sharing parameters globally by all the participating worker nodes. With Bulk Synchronous Parallel (BSP), all the participating workers are designed to send their computed gradients to the parameter server for aggregation at the end of each training iteration. During this iterative process, sensitive model information can be revealed, such as the total number of layers and the size of parameters in each layer. A key enabler of this attack is the need to optimize the synchronization of distributed machine learning systems. Many prior works have verified that the network is the bottleneck in distributed machine learning [5–7]. Modern machine learning platforms that support distributed training, such as MXNet, PyTorch, and TensorFlow, have introduced mechanisms to maximize computation and communication overlaps. Such mechanisms include explicit overlaps between the backward pass calculation and synchronization and implicit overlaps between the forward pass calculation and synchronization. The resulting overlaps are generally at the layer-level granularity of a DNN model. Therefore, an attacker who is monitoring network traffic passively in the system has the opportunity to extract layer-level information.

In the proposed attack, we first seek to identify invariants in network communication through an in-depth study of the design and implementation of distributed machine learning systems. To improve accuracy, we also leverage software-based power side channels to provide more fine-grained information on computation. Such channels are enabled by Intel Running Average Power Limit (RAPL) measurements. Furthermore, we also correlate the temporal and spatial information provided by network and power side channels, respectively. Such correlation helps eliminate noise and anomalies in our collected data. Finally, we also leverage machine learning techniques to generate model profiles for certain operations with high variance. We implemented our attack on the MXNet platform as a demonstrative example. Other platforms also adopt similar computation and communication mechanisms, thus demonstrating similar behaviors. Our proposed approach helps significantly limit the search space of the target architecture of DNN models.

Our main contributions in this work are summarized as following:

– We provide an in-depth analysis of the communication patterns of distributed machine learning and leverage the insight to extract sensitive model information.
– We identify the unique power consumption patterns during model training. We also develop algorithms and systematic approaches to extract these patterns and correlate them with network packets to improve attack accuracy.
– The insight we provide in this work can help improve security in future distributed machine learning systems designs.

2 Motivation

Modern DML frameworks, such as MXNet, Pytorch, and Tensorflow, generally adopt the parameter-server-based architecture to implement distributed training. To coordinate the training process, synchronization is necessary to allow participating workers to aggregate their parameters. During synchronization, workers exchange packets with the parameter server for this purpose. Upon the receipt of the updated parameters, workers would perform local computations for the next iteration. In our investigation, we find that such an iterative process presents high regularity so it is at the risk of information leakage.

Designing neural network architectures is of paramount importance for improving model accuracy. The introduction of deep layers of processing, convolutions, and fully connected layers creates opportunities for various new applications of deep learning neural networks. Many existing efforts have demonstrated remarkable results of neural network model performance improvement by modifying their architectures [8–12]. For instance, Simonyan *et al.* demonstrated that deep convolutional networks can benefit image classification accuracy significantly [10]. Thus, we focus on reconstructing neural network architectures and their hyperparameters in this work.

In the proposed attack, we assume that the adversarial does not need query access to the model either; it does not know its place during the training process. To develop our framework, we make the following assumptions.

Access to the target network. We assume that attackers could access the target network to access the traffic shape or pattern generated during ML training in a public cloud environment. Mehta *et al.* recently studied the network side channel attacks in public IaaS Clouds, and they found that an unprivileged adversary can also indirectly infer the victim's traffic shape by inducing contention with the victim's traffic in a shared network [13]. Also, considering that the network traffic can be noisy, especially in WAN, which helps hide traffic patterns. We also assume that attackers could locate the target machine by identifying the unique communication patterns of DML frameworks.

Co-location. Cloud service providers often assign containers leased by different customers upon the same physical server, where these containers are referred to as co-located containers. It is a persistent threat in the cloud whose feasibility has been demonstrated in several works [14,15]. We assume that attackers could leverage existing techniques to achieve co-residence, thus being able to access the power consumption of the target system. Other existing efforts have also adopted this assumption, such as Cache Telepathy [16] and DeepHammer [17].

While they leverage the cache and RAM side channels, we leverage the power side channel in our study. This work's observations and identified patterns are not specific to any particular DML platform. We use MXNet as an example to implement our prototype framework.

3 Attack Overview

The overall goal of this work is to expose sensitive information and reconstruct DNN models. The sensitive information of interest includes (1) The total number of layers, (2) Layer types, such as fully connected and convolutional layers, (3) Hyper-parameters for each layer, such as the number of neurons, (4) The activation function in each layer, e.g., `relu` and `sigmoid`. Note that the model weights are out of the scope of this paper. The central idea is to collect network and power consumption information via side channels and apply a cross-correlation approach to achieve our goal. Considering the observations mentioned, we design and implement a framework for this purpose.

Architecture Overview. There are three main tasks in the workflow of the proposed framework: forward pass locating, layer identification, and model reconstruction. Each main task consists of small sub-tasks that help achieve their goals. Specifically, forward pass locating is designed to pinpoint the period during which forward propagation is performed. Layer identification helps classify different types of layers of the model. Model reconstruction seeks to reconstruct the original DNN model.

Attack Procedure. First, we map the timestamps obtained from the network trace to the power consumption trace we collect simultaneously. Then we identify the segment where the forward pass is located by extracting the period during which there is no communication between the worker and the server. Due to the implicit communication and computation overlap mechanism of MXNet, the communication of n^{th} iteration may overlap with the computation of $(n+1)^{\text{th}}$ iteration. Also, the *silent period* (defined in Section 4.1) may overlap with the beginning part of the backward pass. Therefore, to determine the exact start time and end time of a forward pass, we need to perform *lookahead* and *lookbehind* of the power consumption trace.

Once we identify the accurate forward pass segment, we then further divide this segment into chunks, each corresponding to the computation of a layer of the target model. This division is based on the DRAM power consumption patterns. We leverage supervised learning techniques to label each power consumption chunk and distinguish different layer types. Generally, each neuron in a layer has an activation process to allow it to learn as per the difference w.r.t error. In this step, we also seek to extract the activation function layer from each power consumption chuck, if available, by leveraging their unique profiles.

In the third step, we perform layer hyper-parameter inference and model reconstruction by leveraging machine learning techniques and domain knowledge. This step mainly relies on network information. Specifically, network information is used to recover the number of layers and the total size of tensors. More details are discussed in the next section.

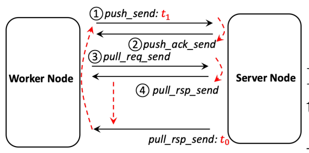

Fig. 1: Communication b/w worker and server

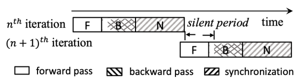

Fig. 2: Communication/computation overlap

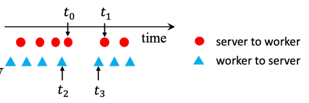

Fig. 3: *Silent Period* identification

Model	Periodicity (secs)
ZFNet	27.889
AlexNet	31.586
MLP-1	11.409
LeNet-5	0.0809
VGG11	8.592
VGG13	10.645
VGG16	14.577

Fig. 4: Network periodicity of different models

4 Forward Pass Locating

In this section, we leverage two key observations in network and power consumption to identify the forward pass locations.

4.1 Analysis of Communication Periodicity

The communication between each individual worker and the server leverages a pair of `push`/`pull` primitives as shown in Figure 1. In this figure, the dashed arrows represent the time sequence, and the solid arrows represent communication directions. Specifically, the worker node initiates the communication by sending `push_send` to send the current parameters to the server node (step ①). Then the server sends back a confirmation message via `push_ack_send` (step ②). Once the worker receives the confirmation, it sends a `pull_req_send` to request the updated parameters from the server (step ③). The server then sends back a `pull_rsp_send` message after the parameters are updated (step ④). This process repeats until the worker receives the last `pull_rsp_send` at time t_0 when synchronization finishes. Note that these message names are only for demonstrative purposes. After step ④, the worker begins the model computations until the updated parameters, such as bias and weights, are ready to be transmitted to the server at time t_1. Intuitively, $t_1 - t_0$ represents the time interval for computation. On the other hand, MXNet parallelizes the backward propagation with synchronization (specifically, step ③ and step ④) to reduce communication overhead. Thus, backward propagation overlaps in time with synchronization, as shown in Figure 2. We can calculate the time interval between the end of the communication of n^{th} iteration and the beginning of the communication of $(n+1)^{\text{th}}$ iteration, and call it *silent period*. This period equals $t_1 - t_0$ in practice. It is evident that the forward pass overlaps in time with this period. We can locate forward pass easily by leveraging this observation.

4.2 Eliminating Outliers

In practice, we can observe many different *silent periods* due to the dynamic nature of network systems and certain specific operations of MXNet. To eliminate these outliers, an intuitive solution is to filter out certain interval groups based on predetermined thresholds. But filtering alone is not enough. This is because certain MXNet operations introduce larger intervals similar to our target period between `pull_req_send` and `pull_rsp_send`. For example, to process `pull_req_send` request, the parameter server invokes `DefaultStorageResponse` function to transfer the model parameters to the worker. In this function, keys must be copied from memory to the response struct. Such operations may result in observable intervals between `pull_req_send` and `pull_rsp_send`.

Therefore, we are motivated to leverage clustering techniques. Essentially, the synchronization can be considered as two independent data streams, packets from the server to the worker and the other way around. The target *silent period* should be among intersections of the intervals in these two data streams. A demonstration is shown in Figure 3, where circles and triangles represent the two data streams, respectively. For clustering, we extract the following features: t_0, t_1, $t_1 - t_0$, t_2, t_3, $t_3 - t_2$, $t_3 - t_0$, $\frac{t_3 - t_0}{t_1 - t_0}$ and $\frac{t_3 - t_0}{t_3 - t_2}$ from the data streams. We also use the sum of squared error (SSE) method to determine the number of clusters, k. The key idea is that as k increases, each cluster's aggregation degree increases, and the SSE value decreases, and vice versa. Thus, we could find the turning point when the decreasing rate of SSE value significantly reduces and identify the corresponding k value. In our analysis, 2 or 3 is typically the optimal number of clusters, which conveniently helps identify the correct *silent period* and outliers. We select the one with the minimum standard deviation values to identify the correct cluster.

To verify that *silent period* is a universal pattern, we select 7 standard DNN models in the image processing domain and leverage Fourier transform to calculate the periodogram of *silent period*, which is an estimate of the spectral density of a signal. Through this calculation, we can estimate the periodicity of this period from tens of and hundreds of iterations of training. The results are shown in Figure 4. Note that *MLP-1* is our custom MLP model with 10 fully connected layers. Each periodicity value represents the average values over three different traces collected from the training of the same model. These values correspond to the time of a complete iteration, which is generally linear with the size of a model given the same training dataset. Note that ZFNet and AlexNet are smaller than VGG16. But they are trained with the Caltech-101 dataset [18], and the other models use the CIFAR-10 dataset [19].

4.3 Identify Exact Timestamps of Forward Pass

Although *silent period* demonstrates a strong correlation with the forward pass, it does not always pinpoint the exact start and end points of a forward pass. The explanation is that MXNet also implicitly overlaps communication and computation [20] by allowing forward pass to begin as soon as the receipt of the first layer parameters. For backward propagation, the first `push_req_send` is generated after the computation begins. This phenomenon is also illustrated in Figure 2. It is challenging to infer the exact start and end times by utilizing network traffic without inspecting the packet payload. Thus, we leverage power trace for this purpose (Power trace collection methods are discussed in detail in Section 5). The specific methods are discussed below.

***Lookahead* Method.** We first map the timestamps obtained from the *silent period* to the power trace we collected simultaneously. This allows us to identify the forward pass's estimated end time (T_e) and an estimated start time (T_s). Then, we perform a *lookahead* method to locate the exact start time T'_s. The motivation behind this method is the observation that the core and DRAM power consumption increase when forward pass calculation happens. A demonstration example is illustrated in Figure 5. This figure shows a segment of DRAM power

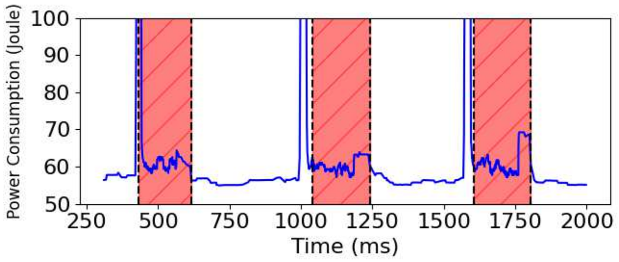

Fig. 5: Overlaps of communication and forward pass

trace collected from the MLP-1 model during a forward pass that overlaps with the *silent period*. The blue curve represents the power consumption of DRAM, where each data point represents the average value within a period of 0.5 ms. Also, only data points whose value exceeds 50 joules are shown in the figure. We can observe that there are three square-shaped power segments whose average values are higher than the rest. They correspond to the three model layers in the forward pass calculation that overlap with *silent period*. Note that the y-axis does not show the entire spectrum of data for better visualization. The power spikes that represent right before the forward pass are caused by memory copy operations. Since they are not persistent for all the models, we cannot leverage them to detect the beginning of a forward pass. Therefore, we develop a sliding-window-based algorithm to detect the square-shaped power segments. The details of the algorithm were omitted due to space limitations. Figure 5, highlighted in red, shows the results of applying this algorithm in the power trace. We can see that this algorithm helps accurately identify the exact start point of the forward pass.

***Lookbehind* Method.** This method helps identify the exact end time T'_e of a forward pass, given the estimated end time T_e. This method also leverages a unique observation in the power consumption trace. It has been known that forward pass execution consumes less memory than backward propagation since there's more memory reuse for forward pass [21]. Therefore, we should observe a significant memory increase upon the start of the backward pass. Such an observation is shown in Figure 6. This segment is generated from the same trace as that of Figure 5. After calculating the moving average, we can observe a steep increase in DRAM power consumption around 3150 ms. To capture this trend, the *lookbehind* algorithm can apply a predefined threshold or use more sophisticated change point detection algorithms [22–24]. The results of the detection are highlighted in red in this figure. Using this method, we can clearly detect the exact start time of the backward pass, i.e., the exact end time of the forward pass.

5 Layer Identification

Once we identify the location in time of the forward pass, we can extract a segment of power trace from the entire trace that corresponds to this forward

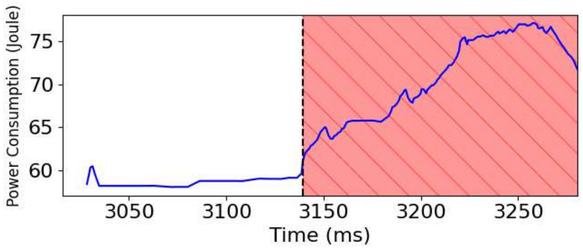

Fig. 6: Overlaps of communication and backward pass

pass execution. Then, we further divide this segment into chunks, each of which corresponds to the computation of a single layer of the target model. This effort mainly relies on the power trace; thus, its accuracy is crucial to the effectiveness of layer segmentation. We discuss our sampling method and evaluate its performance next.

5.1 Power Consumption Sampling

Generally, root privilege is required in order to obtain accurate Intel RAPL counters by reading Model-specific Register (MSR) Linux interfaces. But this would require attackers to perform privilege escalation attacks to access MSR. Alternatively, the Linux power capping framework, *powercap*, provides unprivileged access to the MSRs through the *sysfs* interface. This allows unprivileged attackers to get access to the power consumption data. There are four power domains that are provided by RAPL: package (PKG), power planes (PP0 and PP1), and DRAM. Among all, PP0 provides the power consumption of the cores and DRAM provides that information of DRAM. We mainly rely on these two pieces of information in layer segmentation and layer type classification. A key factor that affects our analysis is the sampling frequency of power consumption data. Without direct access to MSRs, we can sample at most 20000 data points per second. This frequency helps guarantee that we can sample at least one data point for all the layers given that it takes about 70 μs for the smallest layer to execute. Eventually, these data points will be used to detect layer boundaries that we discuss next. Throughout this work, our sample frequency is at the interval of 50 μs.

5.2 Forward Pass Segmentation

To extract each layer from the forward pass, we need to identify the boundary of each layer in the trace. Based on our observation, MXNet saves the state to DRAM, loads up the next layer of the network, and then reloads the data to the system for each layer. We did not test our implementation in GPUs, but one can imagine that this memory operation would be more obvious in GPU given that many high-performance GPU processors have only 1 KB of memory associated with each of the processor cores. Thus, we can observe a DRAM power consumption peak at each layer boundary. This observation can be leveraged to perform forward pass segmentation.

For this operation, instead of using a threshold-based method, we apply a band-pass filter [25] to preprocess the DRAM trace to signify the patterns. Band-pass filters are widely used in wireless transmitters and receivers, whose main function is to limit the bandwidth of the output signal to the band allocated for

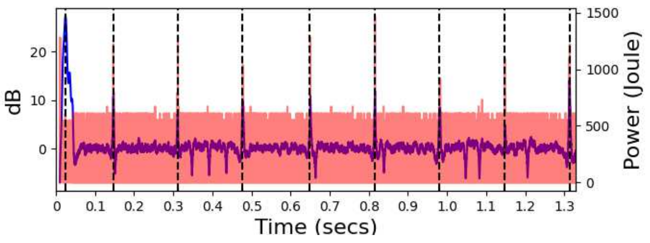

Fig. 7: Forward Pass Segmentation

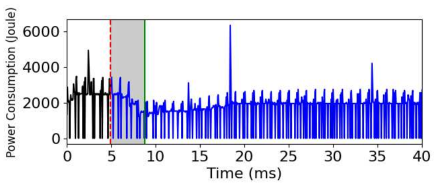

Fig. 8: Core power consumption w/o activation function

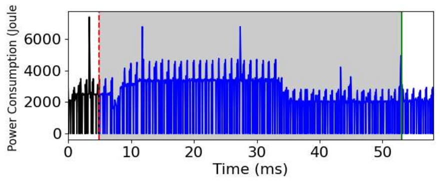

Fig. 9: Core power consumption w/ activation function

the transmission. After this preprocessing, we eliminate noise signals and get a signal containing a band of frequencies, from which we can easily identify the peaks. The processed results of the same MLP-1 model are shown in Figure 7. In this figure, the background data highlighted in red shows the DRAM power consumption, whose values are shown on the right y-axis. The bandpass signal is shown on the left y-axis. The dashed vertical lines in the figure show the identified layer boundaries. Compared with the ground truth boundaries, our identified boundaries have an average error of 2.45 ms, which is about 1.5% of the total duration of a layer. This pattern is universal to all the models.

5.3 Layer Type Classification

Another important piece of information we can derive from the power trace is the different types of each layer in a model.

In this work, we mainly concern two types of layers: convolutional (Conv) and fully connected (FC) layers. Intuitively, each layer should have its unique computation characteristics. For Conv layers, assume the input in is of size $W_i \times H_i \times D_i$ and K_i filters, each of which has dimensions $R_i \times R_i$, then the estimated total amount of Conv layer computation would be in the order of $W_i \times H_i \times D_i \times K_i \times {R_i}^2$. For FC layers, assume the same input of size $W_i \times H_i \times D_i$, and N_i neurons of this layer, then the estimated total amount of FC layer computation would be in the order of $W_i \times H_i \times D_i \times N_i$. By way of rough comparison, we can see that Conv layers are more computationally intensive than FC layers. Such observations have also been verified by Yan *et al.* [16]. But Conv layers are less memory intensive than FC layers due to their partial connections with the neurons of the previous layers and the mechanism of sharing parameters. In addition, there are certain reasonable considerations when designing model architectures as pointed out by Yan *et al.*. For example, the number of filters for Conv layers are typically multiple of 64. These intuitions provide an opportunity for us to characterize and identify different layers.

Table 1: Feature values of Conv and FC layers of the same size

Parameter Sizes		128, 128 × 128 × 9		128, 128 × 256 × 9		256, 256 × 256 × 9		256, 256 × 512 × 9		512, 512 × 512 × 9	
Layer Type		FC	Conv	FC	Conv	FC	Conv	FC	Conv	FC	Conv
Features	T_i (ms)	595.0	250894.7	1098.3	478148.7	1668.3	887469.3	2889.3	1762995.3	15156.7	3130200.3
	$\mathrm{P}_{t_{core}}$	17979	10401666.3	39164.3	20612252	59692.3	38246810	122579.0	77859237.3	331440.3	143407124
	$\mathrm{P}_{t_{dram}}$	755	325573.7	1469.3	646293.3	2121.7	1198849.3	4615.7	2383537.3	9995.7	4708285.7
	$\mathrm{P}_{a_{core}}$	0.7	0.77	0.75	0.74	0.79	0.74	0.63	0.74	1.51	0.66
	$\mathrm{P}_{a_{dram}}$	25.85	31.95	26.7	31.9	28.1	31.9	26.6	32.7	33.3	30.5

Characteristics Comparison of Conv and FC Layers. For this purpose, we extract the following features from the previously obtained power segments and network traffic:

- Total core power consumption ($\mathrm{P}_{t_{core}}$): total core power consumption within the duration of a layer segment.
- Total DRAM power consumption ($\mathrm{P}_{t_{dram}}$): total DRAM power consumption within the duration of a layer segment.
- Average core power consumption ($\mathrm{P}_{a_{core}}$): per-μs core power consumption within a layer segment.
- Average DRAM power consumption ($\mathrm{P}_{a_{dram}}$): per-μs DRAM power consumption within a layer segment.
- Time duration (T_i): the total execution time of layer$_i$.

Then, we leverage these features to perform supervised learning. One might argue that these features are correlated with the number of parameters, thus supervised learning may not be able to make distinctions between these two types of layers of the same parameter size. For verification, we custom-build ten Conv and FC layers of the same parameter sizes, and compare the values of these five features, respectively. The comparison results are shown in Table 1. In this table, the parameter sizes represent the bias size and the total size of weights, which is calculated as $W_i \times H_i \times D_i$. We can observe that the duration, core, and DRAM power consumption of Conv and FC layers differ by at least two orders of magnitude. These numbers also determine that our proposed approach can tolerate a significant amount of forward pass segmentation errors. Although the average core and DRAM power consumption values are similar for Conv and FC layers, they do not affect our classification results. The detailed evaluation results of supervised learning are presented in Section 7.

Activation Functions. In MxNet, activation functions are independent operators connected to Conv or FC layers. Compared to the Conv and FC layers, the computation of activation functions is simpler, thus taking less time. We need to address two key issues: 1. Determine the existence of activation functions. 2. Determine the activation function types, i.e. `relu`, `tanh` or `sigmoid`. In Section 5.2, we associate the peaks of DRAM power consumption with the beginning of the following forward pass calculation because MXNet needs to load the data for the next layer before computation begins. Since DRAM operations happen before the forward pass of the next layer, our timestamps always come a little ahead of the ground truths, thus causing the slight error presented in Section 5.2. Our investigation shows that the CPU is not idle during the time between our timestamps and ground truth – activation functions are being executed if they exist. If not, the CPU becomes idle, leading to a decrease in core power consumption.

An example is shown in Figure 8 and Figuthe re 9. In the figures, the dashed red line denotes the identified layer boundaries with the same technique mentioned in Section 5.2, and the solid green line represents the ground truth timestamp. The figure on the left shows the core power consumption without activation functions. The one on the right shows the scenario with activation function (`sigmoid`) between two model layers. We can leverage the differences in core power consumption to determine the existence of activation functions. Once we locate a DRAM power consumption peak, we can search forward from this point with a sliding window-based algorithm similar to the *Lookahead* algorithm. If a significant increase followed by a decrease in core power consumption is found, an activation function exists; Otherwise, there is no activation function.

For the second issue, we find that the power consumption of an activation function is often positively correlated with the size of the input vector. Given the same parameter size, different activation functions consume different amounts of power. We thus adopt a profiling-based approach to solve this issue. We can get a function $F_a(x)$ for each activation function that estimates power consumption based on parameter size x through offline model training. We use polynomial regression to model $F_a()$. The results of the regression model are shown in Figure 10. In this figure, the x-axis represents the total parameter size $(W_t + B_t)$,

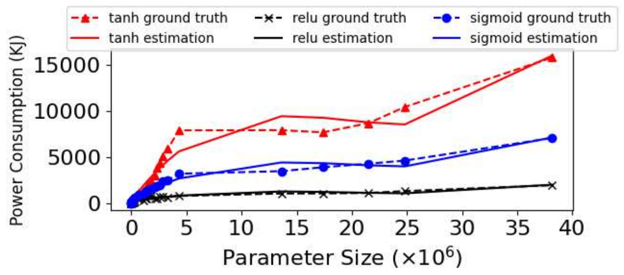

Fig. 10: Polynomial regression

and the y-axis shows the core power consumption values in Kilojoules. This regression is performed offline with ground truth data. We can see that the total power consumption for different activation functions is distinguishable on a given parameter size.

During online attacks, we first reverse the parameter size of each layer (Section 6), decide whether the activation function exists with the sliding-window algorithm, get power consumption estimation with $F_a()$ and parameter size x, then compare this estimated value with the collected power trace to determine which activation function is being used. To determine the period corresponding to the activation function, we use the sliding window as mentioned above-based algorithm to detect the end time point of the activation function.

6 Model Reconstruct

The ultimate goal of adversaries is to steal sensitive model information and use them to reconstruct the same or similar models. To achieve this goal, the adversaries need to obtain the hyper-parameters of each layer and reassemble them together. We call this process hyper-parameters inference (HPI). This effort requires both network and power trace analysis and background knowledge of convolutional layer computation.

6.1 HPI via Network Side Channel

In this subsection, we discuss the techniques leveraged to extract common hyper-parameters of FC and Conv layers.

Number of Layers. In Section 5.2, we demonstrate that the DRAM power consumption pattern during the forward pass computation can be leveraged to perform layer segmentation. Thus, we can identify the number of layers accurately. Alternatively, we can extract this information from the network trace during its overlap with the backward pass computation. This approach is essential because network packets carry payload information, from which we can obtain more accurate estimations of model hyper-parameters. For this purpose, we need to answer two questions.

1. Does communication correlate with the number of layers? To answer this question, we must understand the backward pass computation process. Typically, backward pass computation is made from the last layer backward to the first layer. To improve communication efficiency, MXNet parallelizes backward passes and synchronization. Thus synchronization is performed at layer-level granularity. This unique pattern has also been verified in prior efforts [20]. This allows us to identify the number of layers from the network trace.

Due to the introduction of the subgraph, however, this layer-by-layer communication paradigm will be adapted to subgraph-by-subgraph, which is a coarser granularity level. But we find that MXNet has a warm-up process for initialization, during which the communication is still performed layer-by-layer. This process generally lasts about two iterations. In other words, whether the subgraph is enabled or not, the number of layers correlates with communication during the beginning two iterations of training. Note that such communication patterns are not specific to MXNet. Our following discussions and analysis are conducted on the first two iterations in the network trace.

2. How to extract the number of layers from communication? Intuitively, one might think that packet intervals can identify layer boundaries in the communication of each iteration. However, there can be about 15k intervals whose length is smaller than 20 ms. Such intervals are too small to be identified. Instead of using packet intervals, we find that the packet payload size can be leveraged to achieve this goal. Since backward pass computation is performed layer by layer, when the parameters of the $(n+1)^{\text{th}}$ layer are being transmitted, those of the n^{th} layer are still in computation. Between the gap of parameter transmissions of these two layers, a worker node sends a sequence of 0-payload packets to the server node. We call this sequence Seq_L. The length of Seq_L, i.e. ($|Seq_L|$), depends on the layer size – larger layers result in longer Seq_L. In our observation, the minimum length of Seq_L is 2.

Size of Tensors. In DNN, weights and biases are the learnable parameters exchanged for synchronization in training. All parameters in a neural network are tensors, which essentially are multidimensional arrays that typically have three attributes, *rank*, *shape*, *type*. The *rank* attribute represents the number of dimensions, *shape* represents the number of rows and columns, and *type* is the data type of the tensor's elements, which are generally either 4 bytes or 8 bytes depending on the CPU processors. In our study, we calculate the total size of tensors as rank $\times$ shape $\times$ type, which is the total amount of memory a tensor takes. We attempt to obtain the total size of the weight tensor (W_t) and the total size of the bias tensor (B_t) from the network trace.

A key step in this effort is associating individual packets with each tensor. During synchronization, weight and bias tensors are pushed in order from a worker node to the server node. Therefore, as long as we can identify the separator between the weight tensor and bias tensor, we can calculate the estimated size of the tensors with packet payload size. We design an algorithm for this identification. There are two key indicators of the tensor separators. The first is the `pull_req_send` packets. In the design of MXNet, a worker immediately sends a `pull` request to the server after a tensor finishes transmission via `push_send`. The size of the `pull_req_send` packet payload is 34 bytes. But in practice, ZeroMQ [26] merges small packets into a large ones sometimes. So the packet payload size could also be 100 bytes. It is also possible that the `pull` requests are merged with the `push` packets. In this case, we use the second indicator, which is a sequence of 0-payload packets, called Seq_T. It is essentially the same as Seq_L except that $|Seq_T|$ is always smaller than $|Seq_L|$.

There could be multiple sequences of 0-payload packets (Seq'_T) in our trace, but only one of them is the real Seq_T. To eliminate noise, we use the payload size of the packet that precedes each Seq'_T to determine whether $Seq'_T = Seq_T$. Generally, weights and biases are sent continuously in packet streams. Thus, if the payload size of a packet is smaller than the maximum TCP payload size, it is likely that the transmission finishes. We leverage this intuition to ignore the wrong Seq'_T as shown in line 10 of the algorithm. This algorithm assumes that the weight tensor is transmitted before the bias tensor, which is not always the case in practice. But we can get the correct estimation of each tensor because weight tensors are generally larger than bias tensors. To obtain the accurate number of tensors, i.e., parameter sizes, we can calculate $N_{W_t} = W_t/(\text{data size})$ and $N_{B_t} = B_t/(\text{data size})$ for weight and bias tensors, respectively. The data size is the data type of tensor elements.

6.2 HPI via Computation Processes

To accurately infer the hyper-parameters, we assume the existence of bias in neural network models. The rationale for this assumption is the following. Biases themselves are training parameters that can be used to adjust the feasible region of the algorithm. Further, the impact of geometry on different datasets is different. As a result, data needs to be pre-processed, which would be more challenging with no prior knowledge of the dataset, to counteract the loss of deviation and preserve model accuracy [27]. On the other hand, bias calculation brings little overhead compared to the calculation of weights. Therefore, the bias parameters are essential by providing a cost-effective way for models to improve training accuracy.

FC Layer. A one-to-one mapping exists between the number of biases and the number of neurons in FC layers. Since we can obtain the total size of bias vectors from network trace (Section 6.1), the number of biases, $N_{B_t} = B_t/(\text{data size})$. The data size depends on their data types. Typically, this value aligns with the memory address width of a processor. If each bias is a 32-bit long float number, the data size equals 4; If they are 64-bit long numbers, the data size equals 8. This helps adversaries limit the inferences to a small possible set. Once N_b is known, so is the number of neurons.

Conv Layer. There are four key parameters adversaries need to recover for Conv layers, including the number of filters (N_f), filter shape ($F_w \times F_h$), stride, and padding. To obtain N_f and $F_w \times F_h$, we leverage the following calculation rules of Conv Layers:

$$N_{f_i} = N_{k_i} = B_{t_i}/\text{data size} \tag{1}$$

$$W_{t_i} = N_{k_{i-1}} \times N_{k_i} \times F_{w_i} \times F_{h_i} \tag{2}$$

For Conv layers, both the number of filters (N_f) and the number of kernels (N_k) equal the number of biases, which is described in Eq (1). Note that the subscript i in Eqs. (1) and (2) represents the i^{th} layer. Eq (2) shows how the number of weights can be calculated with the number of kernels of the $(i-1)^{\text{th}}$ and i^{th} layers and the filter shape of the i^{th} layer. Given the above two equations, if W_t and B_t are known, we can easily calculate N_f and $F_w \times F_h$. The specific values of F_w and F_h are typically the same in Conv layers. Thus, they can be calculated by taking the square root value of $F_w \times F_h$. For padding, its size is mainly determined by the filter shape and the stride value. Since stride values generally vary within a reasonably small range (e.g., between 1 and 4), adversaries can generate limited possibilities.

Reconstruct Model Architecture. Once we have all the layer types and parameters, we can assemble them sequentially to reconstruct the model. Note that non-sequential models are out of the scope of our work for now. We can also leverage common practices for basic verification when performing model layer reassembly. For example, FC layers should never come before Conv layers in DNN. Additionally, our proposed reconstruction method is not specific to any learning tasks or datasets.

7 Evaluations

Our evaluations are conducted from three different aspects. First, we evaluate the accuracy of different components in our proposed solution. Second, we perform case studies on two models, MLP-1 and AlexNet, by comparing the reversed model structures and the original ones. We also train the recovered models with the same dataset to compare their accuracy with the original models. Finally, we perform a qualitative analysis on the cost of performing the attack.

7.1 Experiment Setup

We evaluate our work with two Dell OptiPlex-7040 and one Dell OptiPlex-7010 machines, which have 8-core Intel i7-6700 processor with skylake micro-architecture, and 16GB memory. These processors support the execution of Intel RAPL measurements. Among the three machines, two of them are running as the worker nodes, and the other one run as the parameter server and the scheduler. They are interconnected with 1 Gbps network. Our evaluations are tested on Ubuntu 20.04 with MXNet 1.6.0. The ground truth in our evaluations are obtained from application instrumentation and log files. Our evaluations are conducted on 20 different DNN models, including LeNet-1, LeNet-4, LeNet-5, NIN, MLP-1, AlexNet, convNet, *mini*-AlexNet, DeepNet, MLP-2, OverFeat-*accurate*, Overfeat-*fast*, ZFNet, VGG11, VGG13, VGG16, VGG19, VGG11-BN, VGG13-BN, VGG16-BN. All the results presented in this section are average values over three runs.

7.2 Accuracy

For accuracy, we evaluate three perspectives that largely determine the overall accuracy of our reconstructed models.

Accuracy of Layer Type Identification. As discussed in Section 5.3, we utilize the different statistical features of Conv and FC layers to identify their layer types. For this purpose, we explore five supervised learning models, MLP, Naive Bayes, Logistics Regression, Support Vector Machine (SVM), and Decision Trees. To train these models, we collect 41,442 power segments of FC layers and 42,424 power segments of Conv layers. We perform 10-fold cross-validation over the entire dataset and summarize the results in Table 2. For the decision

Table 2: Accuracy of layer type identification

Model	Precision	Recall	F1 score	Accuracy
MLP	0.933	0.979	0.955	0.955
Logistics Regression	0.679	0.854	0.754	0.723
Naive Bayes	0.629	0.977	0.765	0.704
SVM	0.670	0.856	0.743	0.710
Decision Tree (geni)	0.976	0.955	0.965	0.966
Decision Tree (entropy)	0.977	0.975	0.976	0.977

Table 3: Accuracy of layer type identification (unseen models)

Model	Precision	Recall	F1 score	Accuracy
MLP	0.940	0.954	0.947	0.997
DT(gini)	0.967	0.979	0.973	0.998
DT(entropy)	0.969	0.974	0.9717	0.998

tree models, we use *geni impurity* and *entropy* as the criteria for tree splitting, where geni Index $= 1 - \sum_j {p_j}^2$, and entropy $= \sum_j p_j \cdot \log_2 p_j$. From the table, we can see that both decision tree and MLP models perform better than other models. All four criteria, including precision, recall, F1 score, and accuracy can achieve higher than 95%. To avoid overfitting, we limit the maximum depth of the decision tree to be 10, which is selected empirically.

We further test these three models over a separate dataset that is collected from MobileNet0.25, MobileNet0.5, ResNet101_v1, ResNet18_v1, and Resnet34_v1. This dataset contains 14,711 segments of FC layers and 453,118 segments of Conv layers. The evaluation results are shown in Table 3. We can see that MLP and decision tree models still can achieve high precision, recall, and accuracy values. Particularly, the decision tree models can achieve 99%+ accuracy.

Accuracy of Tensor Size Recovery. To evaluate the accuracy of our tensor size recovery mechanism, we compare our recovered tensor size with the original tensor size of all 20 models. In this evaluation, the ground truth is obtained from the `Send()` function invoked by MXNet for parameter synchronization. It contains a data structure that keeps parameter IDs and their corresponding tensor sizes. The average error rate is shown in Table 4. The error rate of each

Table 4: Accuracy of tensor size recovery

Range	500	50331648	100663296	150994944
Error Rate	0.704	0.0148	5.966×10^{-7}	3.179×10^{-7}

layer is calculated as $\frac{|(B'_t + B'_w) - (B_t + B_w)|}{(B_t + B_w)}$, B'_t is the size of our recovered bias tensor, W'_t is the size of our recovered weight tensor, B_t is the size of the original bias tensor and W_t is the size of the original weight tensor. We calculate this error for each layer in our dataset and then present the average error rates. From our results, we find that our mechanism achieves less accuracy when the size of a tensor is small. Thus, we further group these errors into four ranges based on the original tensor sizes. The table shows the maximum value of tensor sizes in each group. Note that each layer contains two tensors, a bias tensor, and a weight tensor.

We can observe that most tensors fall in the range between 500 and 50331648. For the models with more small tensors, such as VGG11-BN, VGG13-BN, and VGG16-BN. Their error rates are not affected much since the size of most of these tensors is close to 500. In practice, adversaries can leverage domain knowledge, such as the size of a tensor is typically a multiple of 64. For example, one of the tensors we collected from LeNet-4 is of size 64 bytes, but our recovery mechanism determines that its size is 105 bytes. In this case, we can generate 2 possible values, 64 and 128, since they are multiples of 64 that are closest to 105. Therefore, the correct tensor size can still be identified by adversaries.

To identify the activation function, we leverage a sliding-window-based algorithm and polynomial regression models for this purpose. We evaluate these two key components separately. The sliding-window-based algorithm is designed to detect the existence of activation functions.

We further evaluate the accuracy of our method in identifying activation function types. The accuracy is calculated based on the condition that the existence of activation functions is validated. In other words, we calculate the percentage of correct function type identifications among all the true positive detection results. The overall identification accuracy is high for most models. But for the VGG models, all of their activation functions are `Relu` functions – their duration is too short to be captured by our method effectively. For instance, we can only sample two power data points during the execution of a `Relu` function in an extreme case. It is difficult to make a decision based on so little information. As a result, many of their activation functions are identified as "unknown" in our evaluations.

In summary, we can identify the layer types and tensor sizes accurately. Our sliding-window-based algorithm can also detect the existence of activation functions accurately. The effectiveness of these components helps lay a strong foundation for our case studies of reconstructing each individual model.

7.3 Case Study

In this section, we use two models, MLP-1 and AlexNet, as examples to demonstrate how our method can be applied in practice for model reconstruction. We also compare the reconstructed models with the original models from two different aspects.

Model Structure Comparison. We compare the reconstructed model structures with the original ones by showing their layer types and layer hyperparameters. The results of the MLP-1 model are shown in Table 5. Since MLP models only consist of fully connected layers, they have three hyperparameters: layer type, tensor (neuron) size, and activation function. Each row in the table shows the hyperparameters of a corresponding layer in the model. It is obvious that we can reconstruct this model structure accurately. The results of AlexNet are shown in Table 6. AlexNet contains both FC and Conv2D layers. For the Conv2D layers, we demonstrate the number of kernels, the kernel shape, and the activation functions. In our tested models, all the kernels have equal width and height, thus using one value for brevity. These layers also have stride and padding parameters, which are not recovered by our proposed methods. But they generally fall within a limited range. AlexNet also has max pooling layers, which contain pool size, stride size, and padding size. Given that they are limited in range, we will discuss the size of the possible set in search space later. Through this comparison, we can see that our method can provide accurate hyperparameter values.

Original	FC, 4096, Relu	FC, 4096, Relu	FC, 4096, Sigmoid	FC, 4096, Relu	FC, 4096, Relu	FC, 4096, Tanh	FC, 4096, Sigmoid	FC, 4096, Sigmoid	FC, 2000, Sigmoid	FC, 1000, Relu
Reconstructed	FC, 4096, Relu	FC, 4096, Sigmoid	FC, 4096, Sigmoid	FC, 4096, Relu	FC, 4096, Relu	FC, 4096, Tanh	FC, 4096, Sigmoid	FC, 4096, Sigmoid	FC, 1984, Sigmoid	FC, 1024, Relu

Table 5: MLP-1 reconstructed structure v.s. original structure

Original	Conv2D, 64, 11, Relu	Conv2D, 192, 5, Relu	Conv2D, 384, 3, Relu	Conv2D, 256, 3, Relu	Conv2D, 256, 3, Relu	FC, 4096, Relu	FC, 4096, Relu	FC, 100, Relu
Reconstructed	Conv2D, 64, 10, Relu	Conv2D, 192, 3, Relu	Conv2D, 384, 3, Relu	Conv2D, 256, 3, Relu	Conv2D, 256, 3, Relu	FC, 4096, Relu	FC, 4096, Relu	FC, 121, Relu

Table 6: AlexNet reconstructed structure v.s. original structure

Original	Conv2D, 96, 11	Conv2D, 256, 5	Conv2D, 512, 3	Conv2D, 1024, 3	Conv2D, 1024, 3	FC, 3072, Relu	FC, 4096, Relu	FC, 1000
Recon-structed	Conv2D, 96, 10,	Conv2D, 256, 4	Conv2D, 512, 3	Conv2D, 1024, 3	Conv2D, 1024, 2	FC, 3072, Relu	FC, 4096, Relu	FC, 1024

Table 7: OverFeat-*fast* reconstructed structure v.s. original structure

Search Space Reduction. Our method has shown promising results in recovering hyper-parameters of FC and Conv layers. This is particularly useful for adversaries to reconstruct models that contain only FC layers, such as MLP models. Given that we might need to find multiples of 64 that are closest to our reconstructed value, we have two possible values for each layer. Therefore, the possible search space for FC layers $= 2^{N_{fc}}$, where N_{fc} represents the total number of layers. For Conv layers, we currently reconstruct max pooling, stride, and padding-related hyperparameters based on our domain knowledge. For example, max pooling layers are generally located between Conv layers, and the padding size is typically within the range of 0 to 2. We can thus calculate an estimated search space for Conv layers $=$ (Stride_Size $\times$ Padding_Size)$^{N_{conv}} \times 2^{N_{conv}-1} \times$ Mp_Size $\times$ Mp_Stride $\times$ Mp_Padding. In this calculation, N_{conv} represents the number of Conv layers, and Mp represents the max pooling Layers. In practice, this possible set could be further reduced. Because certain stride and padding sizes may become unavailable given the input shapes.

7.4 Robustness of the Attack

To evaluate the robustness of the proposed attack, we run another background application, i.e., `Redis` database, that is both communication and memory intensive in a separate container that runs on the same worker node. Specifically, we run a client that issues GET/SET requests to a Redis server by executing `memtier_benchmark` [28]. We vary the memory consumption (between 10% and 20%) and traffic rate of this client by adjusting the read:write ratios, data sizes, the number of concurrent threads, and test time. We then evaluate the impact of this application on two critical steps in the attack: locating forward pass and forward pass segmentation. This experiment is conducted on three models: MLP-1, VGG11, and VGG16. The results are summarized in Table 8.

In this table, *Rate* represents the traffic rate in packets per second (pps). This rate combines the request sending rate and response arrival rate so that it includes the total amount of packets the worker node needs to process per second; *Duration* represents the ground truth forward pass duration in seconds; F_e represents the time difference between our identified forward pass and ground truth; L_e calculates the average error rate of each layer. We define the length of an identified layer i as L_i and the ground truth length as L_{g_i}. Then $L_e = (\sum_{n=1}^{i} \frac{|L_n - L_{g_n}|}{L_{g_n}})/i$. In this experiment, we separately calculate the error rate for the first $i-1$ layers and the last layer.

Table 8: Accuracy of forward pass locating and segmentation

Memory	Model	Rate (pps)	Duration (s)	F_e (ms)	L_e (%)	
					other	last
15%	MLP-1	12030	2.76	1.24	4.72	3.7
	VGG11	9827	4.13	2.64	1.1	15.7
	VGG16	14270	6.59	2.62	2.2	20.3
20%	MLP-1	10389	3.28	1.52	5.89	7.8
	VGG11	9435	5.04	2.92	1.36	20.3
	VGG16	11482	6.94	2.84	3.17	21.3
25%	MLP-1	10120	3.46	2.40	6.2	5.4
	VGG11	9928	5.29	3.44	1.44	23.3
	VGG16	13839	7.26	3.65	4.6	21.6

There are three key observations we can obtain from this table. First, both forward pass locating and segmentation errors increase as the memory consumption and traffic rate increase due to resource contention. Second, for forward pass segmentation, the error rate of the last layer is always higher than that of the other layers. The reason is that the last layers of all three models are FC layers. So it takes them only about 2 ms to finish. Third, despite the interference of the background application, our proposed attack can still achieve high accuracy in detecting forward pass and layer boundaries. In practice, it is not likely to have another co-located background application consuming more than 20% of memory because training complex machine learning models generally requires abundant memory resources.

7.5 Comparison with SOTA

Many similar efforts have been dedicated to obtaining sensitive model information during training. To demonstrate the capabilities of our proposed attack, we make both qualitative and quantitative comparisons with the state-of-the-art techniques, CSI NN [29] and Cache Telepathy [16]. Table 9 shows the information that can be obtained from our work compared to the others.

Table 9: Comparison of attack capabilities

Hyperparameters \ Solutions	Our Work	CSI NN	Cache Telepathy
Number of layers	✓	✓	✓
Layer boundaries	✓	✓	✓
Layer type	✓	N/A	✓
Number of neurons - FC layer	✓	✓	✓
Number of filters - Conv layer	✓	N/A	✓
Filter shape	✓	N/A	✓
Relu identification	✓	✓	✓
Sigmoid identification	✓	✓	✗
Tanh identification	✓	✓	✗
Weights	✗	✓	✗
Strides	✗	N/A	✓
Padding	✗	✗	✓
Max pooling detection	✗	✓	✓
Dropout detection	✗	N/A	N/A
Residue connection detection	✗	✗	✓

In this table, "N/A" means that the work does not mention the corresponding information. Overall, our proposed attack can obtain similar information with other existing work in the literature. However, our proposed attack is built on entirely different assumptions than previous work. For example, CSI NN leverages the timing and electromagnetic (EM) emanations information of an embedded device where an ML model is trained to reconstruct the model architectures, which is not feasible to do on typical server machines. While Cache Telepathy relies on the execution of OpenBLAS and Intel MKL libraries during ML inference, which limits the scope of the attack to specific ML frameworks running on CPUs. In contrast, our proposed attack leverages the inherent execution patterns of ML training that are independent of the utilized hardware and software.

Quantitatively, we also reconstruct VGG16 in our study, which allows us to make a fair comparison with Cache Telepathy. The structure of VGG16 can be fully reconstructed with the techniques proposed in Cache Telepathy. With our proposed approach, we can also fully reconstruct the Conv and FC layers and their corresponding activation functions, number of neurons, and kernel size. But we could not reconstruct the dropout layers and the max pooling layers.

8 Limitations

Although we have shown that network and power side channels reveal sensitive information about models, there are limitations to the type of models we can recover. So far, our solution is limited to sequential models. For non-sequential models, a layer can be connected to multiple layers, resulting in sub-models or blocks. The power and network data we collected do not provide more fine-grained information that allows us to recover the connectivity of layers within a block. Additionally, the mitigation solutions provided by Intel may reduce the attack surface, resulting in potential attack failures. However, attackers can perform this attack from any worker node among all the participating nodes. Given that distributed machine learning or federated learning is gaining popularity among different data stakeholders in various domains, attackers can locate a vulnerable machine to perform the proposed attack.

9 Related Work

Network side-channel attacks are not novel concepts; they have been applied in various domains. Recently, Gu *et al.* proposed a video identification method using network traffic while streaming [30]. Through this attack, adversaries can extract video features. Xu *et al.* proposed a similar effort to infer mobile ABR video adaptation behavior based on packet size and timing information [31]. Network traffic generated by IoT devices has also been leveraged by Apthorpe *et al.* to infer in-home activities [32].

Leveraging side-channel information for DNN model reconstruction has also been explored. Wei *et al.* exploited the GPU side-channel based on context-switching penalties to extract the structural secret of DNN [33]. Power side channels have also been leveraged by Wei *et al.* to reconstruct input images without the knowledge of model parameters [34]. In addition to power, electromagnetic side channels were also utilized to reverse model structures [29]. Hu *et al.* leveraged off-chip memory buses to reconstruct the neural network architectures [35].

10 Conclusion

In this paper, we proposed an effective black-box attack to reconstruct the DNN model architectures. We identified that network communication between a worker node and the server node has unique patterns. We leveraged these patterns to successfully extract the layer hyperparameters, including the number of neurons, the size of bias tensors, and the number of layers. We further correlate the timestamps extracted from network communication with power consumption data collected via software-based power side channels. Such correlation helps reduce the search space of the target model architecture significantly. Through evaluations, we showed that our methods can recover highly accurate hyperparameters. For example, we can identify layer types at 99.9% accuracy with decision tree models. We can also fully recover the size of each tensor of both FC and Conv layers.

References

1. C. A. Choquette-Choo, F. Tramer, N. Carlini, and N. Papernot, “Label-only membership inference attacks,” in *International Conference on Machine Learning.* PMLR, 2021, pp. 1964–1974.
2. J. Jia, A. Salem, M. Backes, Y. Zhang, and N. Z. Gong, “Memguard: Defending against black-box membership inference attacks via adversarial examples,” in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 259–274.
3. M. Juuti, S. Szyller, S. Marchal, and N. Asokan, “Prada: protecting against dnn model stealing attacks,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, 2019, pp. 512–527.
4. M. Kesarwani, B. Mukhoty, V. Arya, and S. Mehta, “Model extraction warning in mlaas paradigm,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 371–380.
5. Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, “Accelerating distributed reinforcement learning with in-switch computing,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 2019, pp. 279–291.
6. L. Mai, G. Li, M. Wagenländer, K. Fertakis, A.-O. Brabete, and P. Pietzuch, “Kungfu: Making training in distributed machine learning adaptive,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 937–954.
7. Q. Ye, Y. Zhou, M. Shi, Y. Sun, and J. Lv, “Dbs: Dynamic batch size for distributed deep neural network training,” *arXiv preprint arXiv:2007.11831*, 2020.
8. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
9. A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.
10. K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
11. F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size,” *arXiv preprint arXiv:1602.07360*, 2016.
12. M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International Conference on Machine Learning.* PMLR, 2019, pp. 6105–6114.
13. A. Mehta, M. Alzayat, R. D. Viti, B. B. Brandenburg, P. Druschel, and D. Garg, “Pacer: Comprehensive network Side-Channel mitigation in the cloud,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
14. D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, “Medusa: Microarchitectural data leakage via automated attack synthesis,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1427–1444.
15. A. Shusterman, A. Agarwal, S. O’Connell, D. Genkin, Y. Oren, and Y. Yarom, “Prime+ probe 1, javascript 0: Overcoming browser-based side-channel defenses,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
16. M. Yan, C. W. Fletcher, and J. Torrellas, “Cache telepathy: Leveraging shared resource attacks to learn DNN architectures,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.

17. F. Yao, A. S. Rakin, and D. Fan, "DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
18. "Caltech101," Apr 2021. [Online]. Available: http://www.vision.caltech.edu/Image_Datasets/Caltech101
19. "CIFAR-10 and CIFAR-100 datasets," Apr 2017, https://www.cs.toronto.edu/~kriz/cifar.html.
20. Y. Xu, D. Dong, W. Xu, and X. Liao, "Sketchdlc: A sketch on distributed deep learning communication via trace capturing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–26, 2019.
21. "Memory Consumption," Mar 2021, https://mxnet.apache.org/versions/1.8.0/api/architecture/note_memory.
22. V. Chandola and R. R. Vatsavai, "A gaussian process based online change detection algorithm for monitoring periodic time series," in *Proceedings of the 2011 SIAM international conference on data mining*. SIAM, 2011, pp. 95–106.
23. Y. Kawahara and M. Sugiyama, "Sequential change-point detection based on direct density-ratio estimation," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 5, no. 2, pp. 114–127, 2012.
24. G. Ristanoski, W. Liu, and J. Bailey, "A time-dependent enhanced support vector machine for time series regression," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 946–954.
25. E. R. Kanasewich, *Time sequence analysis in geophysics*. University of Alberta, 1981.
26. "ZeroMQ," Jul 2021, https://zeromq.org.
27. S. Wang, T. Zhou, and J. Bilmes, "Bias also matters: Bias attribution for deep neural network explanation," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6659–6667.
28. RedisLabs, "memtier_benchmark," 2021. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark
29. L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 515–532.
30. J. Gu, J. Wang, Z. Yu, and K. Shen, "Walls have ears: Traffic-based side-channel attack in video streaming," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1538–1546.
31. S. Xu, S. Sen, and Z. M. Mao, "Csi: inferring mobile abr video adaptation behavior under https and quic," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.
32. N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster, "Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic," *arXiv preprint arXiv:1708.05044*, 2017.
33. J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 125–137.
34. 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*, 2018, pp. 393–406.

35. X. Hu, L. Liang, L. Deng, S. Li, X. Xie, Y. Ji, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "Neural network model extraction attacks in edge devices by hearing architectural hints," *arXiv preprint arXiv:1903.03916*, 2019.