Leaky DNN: Stealing Deep-learning Model Secret with GPU Context-switching Side-channel

Junyi Wei*, Yicheng Zhang†, Zhe Zhou*, Zhou Li† and Mohammad Abdullah Al Faruque†
*Fudan University, Email: wjygerald@gmail.com, zhouzhe@fudan.edu.cn
†University of California, Irvine, Email: {yichez16, zhou.li, alfaruqu}@uci.edu

***Abstract*—Machine learning has been attracting strong interests in recent years. Numerous companies have invested great efforts and resources to develop customized deep-learning models, which are their key intellectual properties. In this work, we investigate to what extent the secret of deep-learning models can be inferred by attackers.**

In particular, we focus on the scenario that a model developer and an adversary share the same GPU when training a Deep Neural Network (DNN) model. We exploit the GPU side-channel based on context-switching penalties. This side-channel allows us to extract the fine-grained structural secret of a DNN model, including its layer composition and hyper-parameters.

Leveraging this side-channel, we developed an attack prototype named `MoSConS`, which applies LSTM-based inference models to identify the structural secret. Our evaluation of `MoSConS` shows the structural information can be accurately recovered. Therefore, we believe new defense mechanisms should be developed to protect training against the GPU side-channel.

***Index Terms*—Deep-learning; GPU; Side-channel;**

I. Introduction

In recent years, technologies driven by machine-learning, especially deep learning, have been gaining strong momentum from the research community and industry. Those technologies have shown promises and early success in transforming the application domains, like computer vision [54] and speech recognition [64]. Driven by this wave, numerous companies have been devoting human and computing resources to develop customized machine-learning models, which have made them **"highly valuable intellectual properties"** [49].

Unfortunately, the high value of machine-learning models also makes them lucrative targets to attackers. Because many machine-learning models are trained on a public cloud or a providing public interface [7], [18], they do present a broad attack surface. As demonstrated by previous works, the parameters (e.g., weights) and hyper-parameters (e.g., regularization terms) of classical machine-learning models like logistic regression can be inferred from the public interface [59], [61].

Stealing DNN model secrets through side-channel. However, applying the attack methods described above against Deep Neural Networks (DNN) models are inefficient for an adversary. Given that DNN models are highly customized and containing a multitude of hyper-parameters, the search space is huge. Recently, a number of works were proposed to steal the model secret through *side-channel attacks* [5], [13], [23]–[25], [41], [63], [65]. Some of those works assume the adversary has physical access to the device so *high-resolution side-channels* about power consumption and accessed memory addresses can be exploited [5], [24], [25], [63]. For the remaining works about the remote adversary, most of them exploited *CPU-based cache side-channel* [13], [23], [65]. Given that the Graphics Processing Unit (GPU) has become the dominant hardware to train and run DNN models, the practical impact of those CPU-based attacks is questionable. The only work investigating GPU was done by Naghibijouybari et al. [41], showing that the number of neurons of DNN's input layer can be learned. Yet, this information provides little guidance for an adversary to recover *the whole DNN structure*. The key question we ask and aim to answer in this work is: **can an adversary infer DNN structural secret like layers and their hyper-parameters by exploiting GPU side-channel?**

As the first step, we revisited the existing GPU side-channel [41] but found it insufficient for our goal. Their attack exploits an Nvidia GPU feature named *Multi-Process Service (MPS)*, which allows the attacker's kernel (called spy) to stay in the same GPU cores with a victim kernel. The spy observes the victim kernel's resource usage by taking samples through CUPTI [45] (Nvidia's performance counters). However, due to the unbalanced scheduling by MPS, the spy is allowed to collect only one sample *at the end of one training iteration*, which is too coarse-grained to reveal the DNN structure.

Comparing to the previous work [41], we pursue the *opposite* direction. We let MPS be switched off (the default setting) and run a spy concurrently with the victim's DNN to force *context switching*. This time, the time-sliced scheduler ensures spy and victim kernels to take fair shares of execution time. Therefore the spy can achieve a much higher sampling rate through CUPTI, as illustrated in Figure 2 and Figure 3.

Challenges. Still, several challenges have to be addressed to recover the model structure. 1) The transition between DNN layer operations (or *op*) is too fast to be observed by the spy, making the ops inseparable from spy's view, due to its insufficient sampling rate. A similar situation exists also for short ops. 2) The execution time for different ops varies significantly, resulting in an uneven number of samples among ops. 3) Different from the kernel co-location side-channel [41] that directly tells spy the victim's resource usage, for context-switching side-channel exploited by us, the only small penalty can be observed, which reflects the victim's resource usage indirectly. In addition, the penalty to the current kernel is

Junyi Wei and Yicheng Zhang are both first author. Zhe Zhou is the corresponding author.

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00031

highly impacted by what has been executed by previous kernels. Discerning ops using this information is more difficult.

Our attack. We address those challenges by developing a hybrid attack framework. To address the issue of insufficient sampling rate, we launch the GPU denial-of-service (DoS) attack using multiple spy kernels to *slow down* the victim kernel. By doing that, a spy is able to obtain a lot more samples per op and increase the prediction accuracy. To address the issue of unbalanced samples and weak side-channel, we design the inference model on top of Long short-term memory (LSTM) model, which is capable of handling complex time-series [31], [36] and utilizing the operation contextual information. In addition, we customize the inference model based on unique insights into DNN training. Instead of identifying layers and hyper-parameters at one pass, we design different LSTM models to identify convolutional ops, non-convolutional ops, and hyper-parameters separately, which increases the prediction accuracy for individual op and hyper-parameter. Due to multi-iteration training, the same execution sequence of DNN layers can be observed many times, which gives the adversary an opportunity to correct misclassification. We develop two *voting* models based on LSTM to merge predictions across iterations. Finally, we leverage the DNN model syntax (i.e., rules well-known to the machine-learning community) to correct the remaining errors.

We implement our attack (named `MoSConS`[1]) based on the above design. By executing `MoSConS`, an adversary will have the capability of inferring the structure of a DNN model *trained* on the cloud by a victim. We found `MoSConS` is effective on the cloud even when the CUPTI access is restricted by the latest NVidia driver [47], after the adversary performs a driver downgrading attack. `MoSConS` is evaluated on a popular GPU (Nvidia GeForce GTX 1080 TI) with TensorFlow installed. Our evaluation shows `MoSConS` can achieve high accuracy for inferring model secret, including **operation sequence, layer hyper-parameters (neuron number, filter size, filter number and stride) and optimizer**. To highlight, after several models are profiled by the adversary, she can predict the layers and hyper-parameters of MLP, ZFNet and VGG16 with 98.4% and 86.6% accuracy on average, which significantly reduces her time, monetary and the labor cost of constructing a full-fledged model. As such, we argue stealing fine-grained model secret is feasible, and we advocate the model secrecy should be considered when building new machine-learning infrastructures, including hardware and system stacks.

Contributions. We summarize our contributions below.

- We developed a new way of exploiting the previously discovered GPU side-channel [41], allowing attacker to inspect another CUDA application at fine-grain.
- We carried out a set of pilot studies to understand how DNN ops are scheduled by system stack and GPU. We also showed how context-switching penalties could be influenced by different spy and victim kernels.
- We developed a new attack `MoSConS` that can extract the structural secret of a DNN model. Our evaluation shows `MoSConS` can achieve high inference accuracy.

[1]Short for Model Secret Extraction with GPU Context Switching.

II. Background

A. Deep Neural Networks

Deep learning is a family of machine-learning methods that feature a transformation from input to output with a cascade of non-linear processing units. A non-linear processing unit is called a *layer*, while the transformation is called a *model*. Before training the model, the developer should define a *model structure*, including what layers to use, their hyper-parameters, and how they should be connected. Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) are the most popular DNNs. CNN cascades layers while RNN features a feed-back loop between layers. In this work, we focus on *CNN* models like other prior works [5], [13], [23]–[25], [41], [63], [65].

Layers. A CNN model typically contains three types of layers: *convolutional* layer, *fully-connected* layer, and *pooling* layer. Each layer consists of a number of *neurons*, which take input from neurons (say $x_1, x_2, ..., x_n$) of the previous layer and computes an output for the next layer. For fully-connected layer, the neuron function is $\phi(\sum_{i=1}^{n} w_i x_i + \beta)$, where w_i is the *weight* for every input neuron x_i, β is the bias value and ϕ is the non-linear transformation function (e.g., `ReLu`). A convolutional layer uses the kernel to filter input so only a spatial region is connected to one neuron. The pooling layer reduces the size of the input to decrease the computational overhead. For example, max-pooling returns the maximum value for a subset of input. We consider the number of layers and layer sequence as a model secret.

Hyper-parameters. Hyper-parameters are set by the developers *before* training. They can be grouped into two categories: the variables specific to each layer (we call them *layer hyper-parameters*) and the variables related to the training algorithm (we call them *model hyper-parameters*). The layer hyper-parameters mainly describe the spatial properties of each layer. The model hyper-parameters include learning rate, number of epochs, batch size, etc. In this work, we focus on *layer hyper-parameters*. In particular, we consider the following as model secret: 1) the activation function used by each layer; 2) the number of neurons for fully-connected layer; 3) filter size of each convolutional layer; 4) the number of filters of each convolutional layer; 5) the stride of each convolutional layer.

DNN training. The primary goal of training a DNN model is to learn neuron weights. Before training, the developer designs a loss function that measures the prediction errors. By gradually minimizing the loss, the developer ultimately gets a satisfying model. The weights adjustment is usually done by a gradient descent optimizer, which calculates the loss first (called *forward propagation*) and then calculates the gradient of the updated loss over weights, from deeper layers to shallower layers (called *back propagation*). With the gradient, each weight is adjusted by subtracting a constant

(learning rate) multiplying the gradient. A training dataset is usually divided into *batches* and the model's internal parameter is updated after all samples of a batch are processed (called an *iteration*). Each run of the training dataset is called an *epoch* and the whole training process typically takes many epochs, consuming days for large datasets [12], [21].

Model structure. While there are only a few basic building blocks for a DNN, the way how they are assembled and tuned has a fundamental impact on DNN's performance. Take image recognition as an example. The evolution of model families from AlexNet [29], VGG [57], Inception [58] to ResNet [22] advances the top-5 performance on ImageNet challenge [54] from 83.6% to 96.43%, which is even on par with human's performance. What's more, even small customization within the same model family can make a big difference. For example, under the VGG model family, VGG19 adds 6 extra 3x3 convolutional layers to VGG13. The error rate can be reduced from 9.6% to 7.5% [57], which can rank it to the second in the ILSVRC2014 benchmark [54]. On the other hand, knowing *which* "knob" to tune and *how* is by no means trivial, as the possible combinations of layers and their hyper-parameters are nearly infinite. As such, the model structure is a key intellectual property [49] and deserves strong protection.

DNN system stacks. To reduce the efforts in training, *DNN system stack* is developed, like TensorFlow [1], Torch [52], MxNet [39] and Caffe [26]. Those system stacks can translate the high-level code (e.g., Python) that describe the model structure to low-level code (e.g., Nvidia CUDA) that are tailored to the hardware platform (e.g., CPU/GPU, single machine/distributed cluster). In this work, we examine our attack against TensorFlow due to its popularity [55].

B. GPU Architecture

Unlike CPU integrating several cores in a die, powerful GPU can have thousands of cores in a single die executing instructions concurrently. Given that DNN operations are mainly based on *Generalized Matrix Multiply (GEMM)*, GPU turns out to be better hardware to train DNN models comparing to CPU. In this work, we focus on Nvidia GPU.

To use the GPU resources for training DNN, Nvidia recommends using its API interface for general-purpose computing called CUDA (Compute Unified Device Architecture) [44]. In particular, a CUDA application (or host application) is composed of a number of *kernels* (i.e., computing tasks) to be executed in GPU. A sequence of kernels that follows execution order is called a GPU *stream*. Each kernel needs to specify the number of *blocks* and how many *threads* to be used by each block [42]. A group of blocks is called a *grid* and all threads under the same block have to be executed in the same SM (Streaming Multiprocessor) and the entirety of a block must be executed before scheduling another block. Inside SM, 32 threads are grouped and launched together (called *warp*) by the *warp scheduler*, which allocates resources per warp (e.g., cores). An SM can have multiple warp schedulers and only one warp is managed by a warp scheduler at a time. Before a warp is scheduled, GPU uses the concurrent scheduler and time-sliced scheduler to determine the execution order of kernels and blocks.

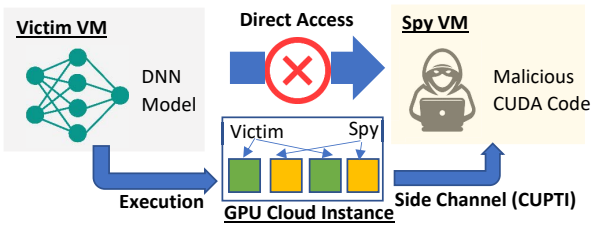

Fig. 1. Adversary model of our attack.

When a CUDA application runs on GPU, a CUDA *context* is created. It is similar to the CPU context, which keeps references to memory, registers, and other state information. When a CUDA application is preempted by another, its context will be switched out and replaced by the next application.

C. GPU Profiler

To help developers of GPU applications with performance tuning, GPU vendors and DNN system stacks offered execution profilers. While they provide valuable insights to developers, we found they can be exploited for model stealing. Below we describe the profilers provided by Nvidia and TensorFlow.

Nvidia profiler. A developer can use *CUDA Profiling Tools Interface (CUPTI)* [45] of Nvidia to profile CUDA applications. In particular, its Event & Metrics APIs allow the developer to interact with the performance counters which log the resource usage of GPU. The developer needs to initialize CUPTI before running the CUDA application. When the application completes, the readings of performance counters will be returned through the Event & Metrics APIs. Take event `L2_subp0_read_tex_hit_sectors` as an example. It represents the number of reading requests from Texture cache that hit the slice 0 of L2 cache. Thus, we can leverage its reading to have an insider look into how the CUDA application interacts with GPU cache and memory.

TensorFlow profiler. TensorFlow provides a *timeline* module [19] to help developers profile the execution of DNN operations on GPU. In particular, it logs the name of each operation, its start and end timestamp and the relevant parameters. To enable the TensorFlow profiler, the developer needs to change a TensorFlow configuration option `trace_level` to `FULL_TRACE`. When the training is finished, the timeline module will keep all the profiled traces into a JSON file which can be visualized by the Chrome browser (loading the JSON file under the page `chrome://tracing`). An example of a timeline is illustrated in Figure 2. In fact, by correlating the CUPTI readings with the TensorFlow timeline, we are able to create a labeled training dataset to build our attack model.

D. Adversary Model

We assume a similar adversary model as Naghibijouybari et al. [41], in which a spy CUDA application (or *spy* for short) and a victim CUDA application (or *victim* for short) are

launched on the same GPU (illustrated in Figure 1). This is feasible in the cloud settings, where multiple applications can share the same physical GPU through I/O pass-through [60], and prior works have demonstrated concrete techniques for machine co-location [4], [53], [67]. We assume those techniques are executed by the adversary.

We assume the adversary is able to run the spy while *training* is performed by the victim. The spy uses CUPTI reading associated with each kernel execution to infer operations run by the victim. The adversary has profiled combinations of layers and their hyper-parameters on the same GPU to be used by the victim. The goal of the adversary is to learn which layers and hyper-parameters are chosen by the victim model. We target the training stage because it typically takes hours or days and the same layer sequence is executed many times [12], [21], leaving abundant opportunities for the spy to extract side-channel information about the victim. Comparing with the work by Naghibijouybari et al. [41], one prominent difference is that our attack does not rely on the activation of Nvidia's Multi-Process Service (MPS) feature. This makes our attack more practical as **MPS is disabled by default** [2].

While Nvidia has released a patch [47] recently to restrict CUPTI access, we found this mitigation can be easily bypassed on the cloud. Based on our test on Amazon EC2, when we rent a VM as the root user, initially CUPTI access is blocked. **But after downgrading the driver version using root's privilege from 418.40.04 (patched) to 384.130 (unpatched), CUPTI can be accessed.** To notice, when spy and victim are on two different VMs sharing the same GPU, the spy can freely downgrade the driver in its own VM, and such action is completely invisible to the victim. Therefore, the GPU side-channel based on CUPTI is still valid. We pick Nvidia as the testing platform due to its high popularity in deep learning. Our tested GPU is from Pascal architecture [46], which was released in 2016 and widely used now.

A number of recent works studied attacks against DNN confidentiality [5], [13], [23]–[25], [41], [63], [65] and they also consider layers and their hyper-parameters as secret. The works are done by Hua et al. [25] and Batina et al. [5] were able to reveal neuron weights but they rely on *physical access* to the DNN accelerator. We will investigate how our adversary (remote) can infer model weights as future work.

III. Understanding Model Execution on GPU

In this section, we report our analysis of how DNN is executed on GPU and motivate the design of `MoSConS`. We first describe how GPU kernels are scheduled in Section III-A. Then, we demonstrate our observation regarding GPU contention in Section III-B. Finally, we show the design of the spy program and the difference between DNN layers' execution traces in Section III-C.

Experiment platform. Our study is carried out on a workstation equipped with Nvidia GeForce GTX 1080 TI. The graphics driver version is 384.11. The CUDA version is V9.0.176 and the cuDNN version is 7.4.2. The workstation has installed Ubuntu 16.04 and we use Tensorflow 1.12.0.

[2]MPS is enabled after a user manually runs `nvidia-cuda-mps-control -d`

A. Scheduling of GPU Kernels

When a DNN model is to be executed, the system stack translates the model structure into the execution plan and the hardware decides how to schedule the computation workload. Unfortunately, not all the details were documented by the stakeholders. Below we summarize the insights we learned through profiling the execution of DNN models.

TensorFlow scheduling. When DNN model is executed on Nvidia GPU, the execution will be carried out by GPU streams and each stream consists of CUDA kernels that invoke Nvidia's cuDNN APIs [11] like `Conv2D` and `BiasAdd`.

We found TensorFlow groups kernels under I/O streams (one stream for CPU-to-GPU data transfer and one for GPU-to-CPU transfer) and compute streams (one or more streams for feed-forward and back-propagation computation). Those streams execute in parallel while the overlap between I/O and compute streams is fairly small (less than 1% of time period when we train VGG [57] and ResNet [22]). Inside each stream, the same kernel sequence is executed under different training iterations. As such, profiling the layer computation under the compute streams and using such information to infer the victim's DNN execution later is feasible. In addition to TensorFlow, we found PyTorch and Caffe schedule GPU similarly (e.g., serializing kernels during training) [26], [51]. As such, our attack is expected to succeed on other stacks too.

GPU Scheduling. When two GPU kernels are executed together (e.g., spy and victim), contention about GPU resources, like cache and atomic memory units, will be introduced. To handle contention, two approaches were developed by Nvidia.

The first approach is to interleave kernel execution and switch context based on *time-sliced scheduler* [9]. A number of time-slices are given to each kernel, and they are scheduled in a mostly round-robin manner. For example, assuming TS_{Ai} and TS_{Bi} are the i-th time-slices for spy and victim kernels, the execution order will be TS_{A1}, TS_{B1}, ..., TS_{Ai}, TS_{Bi}. The duration of each time-slice depends on the priority of the computing task. When a time slice expires, preemption will force context switching between kernels.

The second approach is to let two kernels run concurrently under the *same* GPU context, enabled by a CUDA enhancement named *Multi-Process Service (MPS)* [43]. The MPS service runs like a delegate and let other CUDA applications connect to it and share context, through its scheduler.

B. GPU Contention

The two GPU scheduling approaches introduce a different penalty to host applications and we compare them below.

For scheduling with MPS, Naghibijouybari et al. reverse-engineered the co-location strategy of MPS and showed Left-over policy is adopted such that a kernel takes the idle SM not occupied by the prior kernel [40]. Nonetheless, when both kernels attempt to occupy all SMs (i.e., creating N blocks if

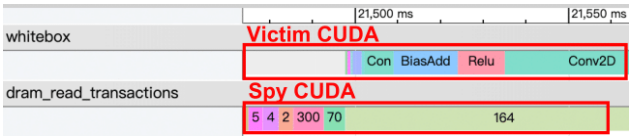

Fig. 2. Spy and victim applications with MPS enabled.

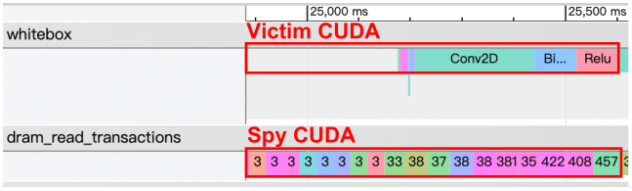

Fig. 3. Spy and victim applications with MPS disabled.

the number of SMs is N), they can achieve co-location *within* an SM. As a result, there is direct contention of the same resources at the same time.

For the time-sliced scheduling without MPS, a kernel does not share context with another kernel. However, context switching between two kernels causes a performance penalty to the following kernel. For instance, when the previous kernel holds a large number of L2 cache entries, higher data-access overhead will be observed for the next kernel.

The work done by Naghibijouybari et al. [41] infers the neuron number of the input layer of a DNN model under the MPS setting, but we found separating the layers and learning their secret is infeasible. Figure 2 and Figure 3 show the CUPTI readings under the TensorFlow timeline with MPS on and off. It turns out MPS increases the execution time of the spy kernel significantly. In most cases, the spy kernel has to wait for a victim's training iteration. As such, when the spy program turns on CUPTI, only one sample can be obtained for one entire iteration. We speculate the slow-down of spy kernel is caused by the uneven resource allocation of MPS service (TensorFlow kernels dominate the resource). On the other hand, time-sliced scheduling is able to make fairer sharing between kernels when they are from different hosts. While Naghibijouybari et al. [41] mentioned that they use "several hundred consecutive kernels in spy" to cover "one whole victim kernel execution", their goal is to ensure co-location of every SM with the victim, rather than obtaining CUPTI readings about different DNN layers. We replicate their setting and found those all kernels finish at the same time.

C. Analysis of DNN operations

Our attack depends on one assumption: different DNN operations result in different context-switching penalty, which is measurable by the spy. Below we test this assumption and report our findings.

Design of spy. We experiment with the spy kernel running 4 blocks and 32 threads to contend with the victim kernels. As such, 4 SMs will be taken by the spy. Given that TensorFlow typically takes all SMs, 4 SMs for spy are enough to observe the penalty caused by context overlapping.

Inside the spy kernel, dummy operations like matrix multiplications are executed to measure the context-switching penalty caused by the prior victim kernel. Table I shows the CUPTI readings about 5 different spy kernels when the victim kernel runs `MatMul` (Matrix Multiplication). It turns out when the spy kernel runs `Conv200` (200x200 convolutional operation), the best result is achieved, with the largest reading on average and the smallest ratio between the standard deviation and the average. In fact, this operation has the largest overlap with DNN operations in terms of the requested units and memory-access patterns, which explains its effectiveness. Another advantage of `Conv200` kernel is that its execution time is shorter (2.5ms) comparing to other kernels, so a higher sampling rate can be achieved.

Impact of DNN operations on spy. `MoSConS` would not work if different DNN operations have same impact on spy's CUPTI readings. To assess the impact, we first test different DNN operations and collect the CUPTI readings with the spy kernel `Conv200`. Table II shows the result, which clearly indicates operation type has different impact on spy.

TABLE I
CUPTI READING OF DIFFERENT SPY KERNELS WHEN `MatMul` IS RUN BY VICTIM. EVENT_1 IS **FB_SUBP1_WRITE_SECTORS**. EVENT_2 IS **FB_SUBP0_READ_SECTORS**. WE RUN EACH COMBINATION OF SPY AND VICTIM FOR 10,000 TIMES. THE RESULT IS REPRESENTED AS "AVERAGE (STANDARD DEVIATION)".

Spy Kernel	Event_1	Event_2
VectorAdd	2.64(18.57)	164.50(68.28)
VectorMul	2.44(2.56)	163.39(22.94)
MatMul	15.94(7.51)	1472.27(82.92)
Conv100	629.51(115.23)	1548.48(256.18)
Conv200	2525.85(16.59)	2489.24(175.62)

TABLE II
CUPTI READINGS OF `Conv200` SPY KERNEL WHEN DIFFERENT VICTIM OPERATIONS ARE EXECUTED. `NOP` MEANS THE VICTIM KERNEL IS IDLE. THE RESULT IS REPRESENTED SIMILARLY TO TABLE I.

Victim Op	Event_1	Event_2
MatMul	2525.85(16.59)	2489.24(175.62)
Conv2D	2542.45(28.73)	5695.03(976.65)
ReLU	0(0)	1164.95(1076.20)
BiasAdd	0(0)	948.89(777.74)
Sigmoid	0(0)	1287.55(1017.06)
NOP	34943.46(232.18)	18454.04(5370.17)

IV. Model Extraction Attack

Challenges. Given that DNN structure can be seen as a sequence of operations (or *ops* for short) and different ops result in different samples read by a spy, a natural solution for spy would be separating the sequence of samples first and then identifying the individual op from each segmentation. This approach is widely used for side-channel attacks when the secret is a sequence, e.g., PIN inputted on smartwatch [33], [62]. However, this approach does not fit our problem for three reasons. 1) spy's sampling rate is relatively low (less than 1K samples per second) comparing to other powerful side-channels (e.g., 2 Giga samples per second for power

analysis [38]), making **op transitions and short ops undetectable**. 2) computation-intensive ops like `Conv2D` take much longer time to compute, resulting in uneven samples among ops. 3) the CUPTI readings fluctuate from time to time even for the same op (e.g., the standard deviation of `ReLU` is 1076.20 shown in Table II). Those observations indicate the complex relations between DNN structure and the context-switching side-channel, which cannot be addressed by simple statistical or machine-learning models. We tackle those issues by launching a novel *"slow-down"* attack first and then using *LSTM (Long short-term memory)* models to detect different DNN ops and hyper-parameters.

Slow-down attack on victim kernels. Though by shortening the execution time of a spy kernel we can collect more samples, we found the victim ops are less distinguishable. Therefore, we choose to *elongate the victim 's execution time* to obtain more samples. Our approach is to let the attacker launch more kernels inside the spy program, through which the time-sliced scheduler will reduce the duration of each time-slice for victim.

In particular, we tested hundreds of kernel parameter combinations by changing <#kernels, #blocks, #thread>, with the value of each ranging from 1 to 32 (multiplied by 2). We found there is an upper-bound of the slow-down ratio, such that higher numbers of kernels/blocks/threads are not always more effective. After the above analysis, we decide to use 8 kernels for spy and put 2 kernels into a group with the same settings. Assume each group is named as G_i while $0 \leq i \leq 3$. The number of blocks and threads for each kernel are set to 4×2^i and $4 \times 2^i \times 32$. Our empirical result shows the victim can be slowed down for 17 times while the spy is slowed down for less than 3 times. To notice, we can run the slow-down attack sparingly so the overall victim training time will be increased only slightly. In Section V-F, we elaborate on the performance overhead of slow-down attacks on the victim.

Overview of the inference attack. Before the actual attack, the adversary profiles models under different families to train the inference models. During the attack, the adversary waits for the bootstrapping of a TensorFlow process (e.g., through monitoring GPU usage) before launching the inference attack. Once training is started, the spy kernels are launched. When enough CUPTI readings are collected, all spy's kernels will be terminated. Next, the spy's CUPTI readings will be processed by the inference models to predict the op sequence of the victim, which we call `OpSeq` for short. The hyper-parameters will be inferred as well and attached to `OpSeq`.

We found the unique characteristics of DNN training can be exploited to develop sub-models for inferring layers and hyper-parameters separately with high accuracy. 1) A spy can obtain multiple samples for convolutional ops (`conv` for short) and matrix multiplication ops (`MatMul`) due to their long execution duration. 2) DNN training usually takes many iterations resulting in the repeated `OpSeq` of the victim. Between iterations, there is a gap where no kernel is executed. 3) Similar to natural-language sentences, DNN model has

TABLE III
STRUCTURES OF THE 5 LSTM MODELS USED IN MoSConS.

M_{long}	V_{long}	M_{op}	V_{op}	M_{hp}
LSTM 256 LSTM 256				LSTM 128
FC				
Softmax				
Cross-Entropy				
Weighted Sum	Sum	Sumif		Sumif

"syntax" about the order of ops and hyper-parameters (e.g., pooling should presume `conv`). Based on those insights, we develop 5 LSTM models and one simple machine-learning model (M_{gap}) for different tasks and assemble their results. Below we overview the attack flow (also illustrated in Figure 4). The structures of all LSTM models are shown in Table III.

- **Splitting iterations.** After the attacker obtains CUPTI samples, she splits them based on the gap between consecutive iterations using an inference model M_{gap}.
- **Recognizing long ops.** The attacker looks into the samples within one iteration, and classifies each one into `conv`, `MatMul` or `OtherOp` using M_{long}. Then, the attacker combines the predicted ops across all iterations to correct the misclassified ones, in a process called *voting* using V_{long}.
- **Recognizing other ops.** For other layer ops (e.g., `MaxPool`) or activation functions (e.g., `ReLU`), the attacker classifies them using M_{op}. The attacker uses V_{op} to vote and refine the classification result. `OpSeq` is generated by combining all inferred ops.
- **Inferring hyper-parameters.** The attacker uses M_{hp} to infer hyper-parameters related to ops and the whole model (e.g., optimizer).
- **Model correction.** After the above steps, the adversary corrects ops and hyper-parameters according to DNN syntax.

After the above steps, the attacker can reconstruct the victim's model structure. In most cases,combinations of consecutive ops can be deterministically mapped to layers. For example, a convolutional layer runs `conv` and `BiasAdd`, while a fully-connected layer runs `MatMul` and `BiasAdd`. However, when non-sequential connection between layers are introduced, like shortcut used in ResNet [22], `OpSeq` could reflect multiple model structures. We discuss this issue in Section IV-C.

Selecting CUPTI counters. While Nvidia provides tens of CUPTI counters, using all of them for `MoSConS` does not lead to an optimal result. Firstly, some of them are always zero or constant, revealing nothing about the victim. Secondly, using more counters will increase the execution time of the spy kernel, reducing the sampling rate. By checking the documentation, we found the counters are divided into multiple groups and the execution time of a spy kernel depends on the number of groups it access. In the end, we select 3 groups of

10 counters relevant to convolution or matrix multiplication operations and they are listed in Table IV.

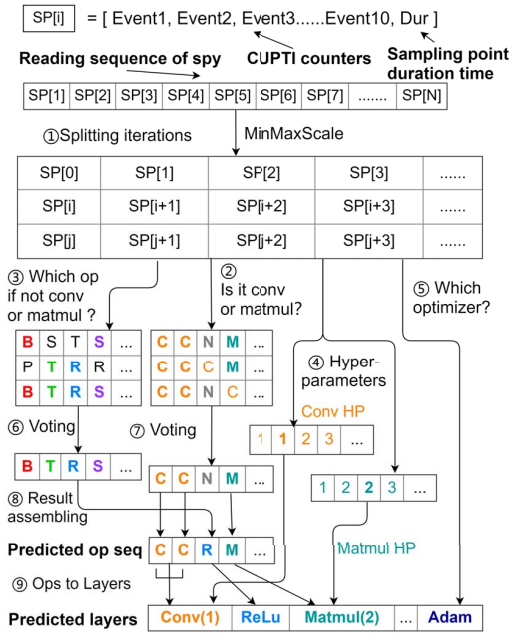

Fig. 4. Overview of attack flow. The sample sequence of spy is split to multiple iterations ①. Then they are sent to M_{long}, M_{op} and M_{hp} models to predict the victim ops (e.g., N at ② and `(R)elu`, `(P)ooling`, `(M)atMul` and `(B)iasAdd` at ③), hyper-parameters ④ and optimizer⑤. Voting is done by V_{long} ⑦ and V_{op} ⑥ to correct mis-predicted ops. Finally, the victim's model structure is revealed after collapsing consecutive ops and correction with model syntax (⑧ ⑨).

A. Splitting Iterations

We choose to split the iterations first because 1) voting across layers cannot be achieved otherwise; 2) the gaps between consecutive iterations are easier to identify.

We develop a model (called M_{gap}) based on Light Gradient Boosting Machine (LightGBM) [37] to classify each sample of spy into `NOP`[3] or `BUSY`. The input to the LightGBM is a vector of CUPTI samples and each sample contains values from the counters (all integer values) described in Table IV. Before classification, we also apply MinMaxScale to pre-process the input and convert each feature value to be ranged from 0 to 1, in order to prevent training bias.

While the `NOP` sample is usually associated with the iteration gap, we found sometimes it also exists within a layer. On the other hand, the gap usually contains multiple `NOP`. As such, we set a threshold TH_{gap} and split iterations if the number of consecutive `NOP` is above TH_{gap}. Before moving to the next step, we also need to identify the iterations with invalid sample sequence and remove them. Those iterations emerge because of incomplete sampling, e.g., the spy is launched in the middle of a training iteration. To remove them, we compare the number of samples to the average across iterations. Assuming the average number of samples is RC_{avg} and the lower-bound and upper-bound threshold ratios are R_{min} and R_{max}, the samples of a valid iteration should be within $[RC_{avg} * R_{min}, RC_{avg} * R_{max}]$. We elaborate on the parameter choices in Section V-A.

[3]We use `NOP` for simpler presentation. It does not mean that the GPU is executing a `NOP` instruction.

B. Predicting `OpSeq`

Recognizing long ops. Spy's primary goal is to identify convolutional layers and fully-connected layers, because they have the biggest impact on the performance of a CNN model. During feed-forward training, a convolutional layer sequentially invokes `conv` (e.g., `Conv2D`), `BiasAdd` and an activation op (e.g., `ReLU`). During back-propagation, it calculates the gradient in a reverse order using ops according to the feed-forward pass (e.g., `ReLUgrad`, `BiasAddGrad` and `Conv2DBackprop`). Fully-connected layer executes `MatMul`, `BiasAdd`, and activation op like `ReLU` in feed-forward training. Similarly to the convolutional layer, the reverse of the feed-forward ops will be executed during back-propagation (e.g., `ReLUgrad`, `BiasAddGrad`, and `MatMul`).

Among those ops, `conv` and `MatMul` are easier to identify because of their long execution time. We develop M_{long}, an LSTM model to classify each CUPTI sample into four classes: `conv`, `MatMul`, `OtherOp` and `NOP`. LSTM is used for this task because it has shown many successes in handling complex time-series [31], [36] and can remember features over the long term or short term.

Our LSTM model takes a vector of the selected counters under a CUPTI sample as word input and outputs a four-dimension vector, which contains the logit values of the 4 targeted classes. When training the LSTM model, we found the number of samples for `conv` differs a lot from other ops, leading to a very imbalanced training dataset. As a result, using the samples for training under the default LSTM settings would not yield good accuracy. To address this issue, we designed *weighted softmax* and customized *cross-entropy loss* to compensate for the imbalanced data. Specifically, when calculating the loss for a sample, we calculate the softmax and cross-entropy loss between the logits outputted by the LSTM model and the ground-truth labels (the op names of TensorFlow) processed by one-hot encoding. Then, the loss is amplified by a constant if the sample is from the minor class.

Voting. DNN training usually takes many iterations and therefore the spy can obtain multiple predicted `OpSeq`s, which can be combined to correct the erroneous predictions at individual sequence. As such, we designed a voting procedure for error correction.

Our voting model, named V_{long}, is developed also based on LSTM. The model consumes the output from M_{long} over all iterations to correct the mispredictions. The model is trained with standard softmax and cross-entropy loss against ground-truth. Specifically, when the attacker monitored n iterations, the input would be a $4 * n$ dimension vector where each 4-dimension word is a one-hot prediction from one iteration.

To be noticed is that the predicted `OpSeq`s across iterations need not be aligned before being inputted to the voting model

TABLE IV
SELECTED CUPTI EVENTS & METRICS COUNTERS.

Group(#counters)	Counter	Description
1(2)	Tex0/1_cache_sector_queries	Number of texture cache 0/1 requests..
2(4)	Fb_subp0/1_read/write_sectors	Number of DRAM read/write requests to sub partition 0/1.
3(4)	L2_subp0/1_write/read_sector_misses	Number of write/read requests sent to DRAM from slice 0/1 of L2 cache.

as LSTM is capable of memorizing values over arbitrary time intervals. Without losing generality, we choose the timeline of the first iteration's `OpSeq` as the base timeline, and let other `OpSeq`s be compared with it.

Recognizing other ops (`OtherOps`). The rest ops of convolutional and fully-connected layer, as well as ops of other layers, are recognized by another inference model named M_{op}. We aim to detect `BiasAdd`, activation functions like `ReLU` and their back-propagation versions (e.g., `BiasAddGrad` and `ReLUgrad`) for convolutional and fully-connected layers. For pooling layers, only one op is executed for feed-forward (e.g., `MaxPool`) and back-propagation passes (e.g., `MaxPoolGrad`). We call those ops as `OtherOp`. M_{op} takes the same input as M_{long} and outputs class logits of `OtherOp`. It also uses the softmax and cross-entropy loss. But when calculating the total loss, it does not sum up the loss from all samples. Instead, it only counts in those relevant to `OtherOp`. In other words, the loss resulted from `Conv2D`, `Conv2DBackprop` and `NOP` samples are all neglected. The prediction results are refined by a voting model named V_{op}, which is almost the same as V_{long}. Similar to the step above, when training the voting model, only the loss from `OtherOp` are counted.

To notice, M_{op} still makes predictions for those samples that are irrelevant to `OtherOp`, like the back-propagation ops. Those predictions though are not used by the attacker but they are implicitly used by the LSTM model to make predictions for the subsequent sample, as the LSTM model can memorize the input status.

Collapsing ops. For long ops like `conv`, multiple samples can be observed, leading to multiple predictions. We collapse the consecutive predictions if they are identical into one op. For example, if the predicted `OpSeq` is {`ReLU`, `conv`, `conv`, `conv`, `BiasAdd`}, the new `OpSeq` after collapsing will be {`ReLU`, `conv`, `BiasAdd`}.

C. Inferring Hyper-parameters

In addition to learning ops, the attacker also needs to learn the layer hyper-parameters, like filter size as stated in Section II-A. We designed an LSTM model named M_{hp} to predict them. The model has the same structure as M_{op}, but the training data is constructed differently. For neuron numbers, the label is assigned to the last sample of a convolutional or fully-connected layer. We use the last sample because it encourages M_{hp} to make full use of the information from all the samples related to the layer.

Other hyper-parameters. `MoSConS` reveals most of the hyper-parameters uncovered by previous works, even though we assume a much weaker adversary (remote attacker without physical access). On the other hand, the difficulties of uncovering all hyper-parameters are not the same. For example, shortcut [22], a technique to add a connection between non-adjacent layers, is not recovered by `MoSConS`. The model developer can choose where to place the shortcut but such information is not visible to the spy with only CUPTI access.

However, the attacker can leverage the domain knowledge to infer the places of shortcuts after the layers are learnt. For example, if the layer structure is similar to ResNet [22], the shortcut is likely to bypass every 2 convolutional layers.

D. Correction with Model Syntax

After the previous steps, the attacker will obtain an `OpSeq` augmented with hyper-parameters but errors exist due to imperfect prediction. On the other hand, we found the attacker can correct the predicted `OpSeq` with heuristics pertinent to DNN syntax. Firstly, certain ops have inherent dependencies and the wrong order will break the training process. For example, in the feed-forwarding phase, `conv` and `MatMul` are always followed by a `BiasAdd` and an activation op like `ReLU`. Secondly, the layer hyper-parameters depend on the ops of the current and adjacent layers. For example, the number of filters for a convolutional layer or the number of neurons for a fully-connected layer usually gets doubled when the layers go deeper. Besides, they are set to the power of two as choosing other values often leads to inferior model performance. Therefore, the attacker can correct the wrong hyper-parameters based on these rules.

While this step requires the attacker's domain knowledge in DNN, we want to point out those heuristics are well known. In addition, as shown in our evaluation, previous steps already produce high-quality `OpSeq` and hyper-parameters. In fact, only a few predictions need to be corrected.

V. EVALUATION

In this section, we evaluate the inference accuracy of `MoSConS`. Firstly, we introduce the settings for our evaluation, including the testing procedure, the parameter settings, the dataset, and the evaluation metrics. Then, we describe the evaluation result, focusing on `OpSeq` and hyper-parameters.

A. Experiment Settings

Experiment procedure. Our experiment consists of two steps. Firstly, we use the spy program to profile a set of CNNs and train the inference models. Each spy's CUPTI reading is attached with ground-truth values (e.g., `conv` and filter size 3x3). We achieve this goal by aligning the model's ops with spy's readings using the TensorFlow timeline profiler. Sometimes, a spy kernel can overlap with multiple victim ops on the timeline and we choose the TensorFlow label having

TABLE V
THE LAYERS AND HYPER-PARAMETERS OF PROFILED MODELS. THE SUBSCRIPTS ARE HYPER-PARAMETERS, INCLUDING THE NUMBER OF NEURONS, FILTER SIZE, NUMBER OF FILTERS, STRIDE AND ACTIVATION FUNCTIONS. THE LETTERS ARE EXPLAINED IN TABLE VII.

Model	Structure
Cust. MLP	$M_{64,R} - M_{128,T} - M_{256,S} - M_{512,R} - M_{1024,T} - M_{2048,S} - M_{4096,R} - M_{8192,R} - M_{16384,S} - Optimizer_{Adagrad}$
AlexNet	$C_{11,96,4,R} - P - C_{5,256,1,R} - P - C_{3,384,1,R} - C_{3,384,1,R} - C_{3,256,1,R} - P - M_{4096,R} - M_{4096,R} - M_{1000,R} - Optimizer_{Adam}$
Cust. VGG19	$C_{13,64,1,R} - C_{13,64,1,R} - P - C_{11,192,1,R} - C_{9,256,1,R} - P - C_{7,256,1,R} - C_{5,256,1,R} - C_{3,256,1,R} - C_{1,256,1,R} - P - C_{3,512,1,R} - C_{3,512,1,R} - C_{3,512,1,R} - C_{3,512,1,R} - P - C_{1,512,1,R} - C_{1,1024,1,R} - C_{1,2048,1,R} - C_{1,4096,1,R} - P - M_{4096,R} - M_{4096,R} - M_{1000,R} - Optimizer_{GD}$

the largest overlap with the spy kernel. Secondly, we run a different set of models and use the trained inference model to reveal the model secret.

For the models to be profiled, we choose MLP (Multilayer Perceptron), AlexNet [29] and VGG19 [57]. They cover most of the CNN building blocks. Table V lists their structures.

For the models to be tested, we choose a different MLP (five fully-connected layers), ZFNet [66] and VGG16. By testing this combination, we evaluate the effectiveness of `MoSConS` against the victim's model customization under the same family and **across families** (AlexNet and ZFNet are different families). Table IX shows the structures of tested models.

We feed images from ImageNet [54] to the profiled and tested models to simulate the normal training workload. The original image size is 64x64 and we convert the size to 224x224, which is a standard pre-processing technique used by model developers to smooth the gradient [8]. We randomly choose 10,000 images to train each model. The batch size is set to 64 for VGG19 and VGG16, 512 for AlexNet, 256 for ZFNet, and 128 for MLP and customized MLP. Each model is trained for 500 iterations.

Parameter settings. We use a few pre-defined parameters for iteration splitting based on empirical analysis. In particular, we set TH_{gap} (minimum number of NOP per gap) to 6, R_{min} (minimum ratio of operations for an iteration) to 0.8, and R_{max} (maximum ratio) to 1.2. To notice, those numbers can be adjusted based on the targeted GPU.

Evaluation metrics. We consider the accuracy as the main metrics to evaluate the prediction. As an example, for `conv` ops, we assume there are N `conv` ops in the victim model, and their positions on the ground-truth `OpSeq` are $L =< l_1, l_2, ..., l_N >$. Attacker speculates there are M `conv` ops and P of them correctly match L. The accuracy is $\frac{P}{N}$.

B. Iteration Splitting

This step needs to achieve high accuracy to ensure the data for the later stages are usable. We tested this step using the same training and testing models. We count the number of `NOP`, ops and the correctly identified ones for computing accuracy. Table VI shows the result. As we can see, the accuracy is quite high (all over 94%). In addition, we found

TABLE VI
RESULT FOR ITERATION SPLITTING. CUST. IS SHORT FOR CUSTOMIZED.

Model	Op	# Ops (Accuracy)
Cust. MLP	NOP	203,721 (99.201%)
	BUSY	204,914 (99.139%)
ZFNet	NOP	716,600 (94.687%)
	BUSY	470,346 (96.316%)
VGG16	NOP	721,117 (96.456%)
	BUSY	514,180 (94.084%)

our heuristics based on TH_{gap}, R_{min} and R_{max} are quite effective to identify gaps between iterations. We rarely detect gaps as false positives. Achieving high accuracy at this task also helps us to obtain many "clean" iterations (covers the entire `OpSeq`) for the voting task. We use 5 iterations for voting. Besides, we also evaluated different batch (16 to 512) and image size (32 to 384) and found their impact is quite small: on VGG16, the accuracy of identifying `NOP` ranges from 96% to 98%.

C. Op Inference

Table VII shows the accuracy of this step. The number of predicted ops are counted before op collapsing. The accuracy for customized MLP, ZFNet and VGG16 are in average 90%.

The effectiveness of `MoSConS` is lower for VGG16 so we look into its false predictions. We found that most errors occur at ops like `BiasAdd` and activation, which is less important than `conv`. These errors are also easy to correct because they show up together at a fixed pattern within the entire `OpSeq`. For example, a model usually uses the same type of activation function. All `conv` are followed by `BiasAdd`. Given that our inference model rarely makes mistakes on key ops like `conv` and `MatMul`, the attacker can correct those misclassified ops easily. Also, we found there are alignment issues of ops when reaching the end of `OpSeq`. Such issues can be partially resolved with op collapsing and syntax correction.

D. Hyper-parameter Inference

We evaluated the prediction accuracy for hyper-parameters. We are able to cover 5 of the 6 sensitive hyper-parameters defined in Section II. Because some hyper-parameters under the tested models have constant or limited values, we vary those hyper-parameters on the profiled and tested models just for this evaluation step. Below are the test settings.

- **Filter size.** We change the filter size of VGG19 and AlexNet to 7 different value (1x1, 3x3, 5x5, 7x7, 9x9, 11x11 and 13x13) under different layers and predict if those values can be correctly identified.
- **Number of filters.** We vary the number of filters from 64 to 4096, multiplying 2 on VGG19 and AlexNet.
- **Number of neurons.** We vary the number of neurons from 64 to 16384, multiplying by 2 on a customized MLP.
- **Stride.** We change the stride size from 1 to 4 on VGG19 and AlexNet.
- **Optimizer.** As the optimizer is implemented as a cuDNN op, we evaluated whether this can be predicted as well

TABLE VII
ACCURACY FOR OP INFERENCE (C=CONV, B=BIASADD, R=RELU, P=POOLING, M=MATMUL, T=TANH, S=SIGMOID). PRE V. AND W/ VT. ARE SHORT FOR PRE-VOTING AND WITH VOTING.

Model	Phase	C	M	B	P	R	T	S	Overall
Cust. MLP	Pre Vt.	-	97%	-	-	98%	98%	98%	97.05%
	W/ Vt.	-	99%	-	-	100%	97%	100%	99.38%
ZFNet	Pre Vt.	95%	87%	98%	92%	54%	-	-	86.25%
	W/ Vt	97%	98%	88%	86%	91%	-	-	92.96%
VGG 16	Pre Vt.	82%	99%	77%	73%	88%	-	-	84.75%
	W/ Vt.	86%	100%	87%	77%	83%	-	-	85.81%

TABLE VIII
OVERALL ACCURACY FOR HYPER-PARAMETER PREDICTION.HP1, HP2, HP3, HP4, HP5 REFER TO THE NUMBER OF FILTERS, FILTER SIZE, THE NUMBER OF NEURONS, STRIDE AND OPTIMIZER.

Hyper-parameters	HP1	HP2	HP3	HP4	HP5
Accuracy(%)	95.71	88.1	96.58	95.89	92.63

(regarded as model hyper-parameter). Three optimizers (Adagrad, Adam and GD) are tested.

After the different hyper-parameter values are profiled to train M_{hp}, we test M_{hp} on our three tested models. Table VIII shows the accuracy, ranging from 88.1% to 95.89%. The accuracy of filter size is lower, mainly because different values have little impact on op's execution time, making the observed readings more indistinguishable.

E. Layer Sequence Inference

We evaluate to what extent attackers can recover the whole layer sequence (including hyper-parameters) with the right order. In this case, continuous identical ops are collapsed to one op and layers can be derived based on op combinations. As such, even there are misaligned ops causing errors in op inference (e.g., ground-truth and the predicted sequence are `CCBR` and `CCCBR`, resulting in 3 misclassified ops), those errors can be corrected easily. Table IX shows the results, we listed 2 rows (ground-truth and predicted structure) for each model.

We took a closer look at the false predictions and found they are usually caused by ops with very short execution time. Specifically, a VGG16 training iteration (the feed-forward phase) lasts around 7000 ms, consisting of 130 ops. On average, an op lasts 54 ms (7000/130). In contrast, each spy kernel only lasts for 16-19ms. Therefore by expectation, the attacker can sample each op three times. Nonetheless, the 10 shortest ops last less than 5 ms, meaning that multiple layers would be squeezed in one spy sample. Fortunately, those short ops are usually `BiasAdd` or activation functions like `ReLu` that are less critical than `conv`. The layers can still be correctly identified even when those short ops are misclassified. We apply model syntax to edit some incorrect layers, and the final results are shown in the "Predicted structure" row. `MoSConS` is able to achieve 100% accuracy following the right layer sequence for customized MLP and ZFNet, 95.2% for VGG16. For hyper-parameter prediction, the accuracy is 100%, 76.9%, and 82.8%.

F. Performance Impact of the Attack.

We force GPU context-switching by running spy concurrently with the victim. As a result, the performance of the victim DNN is expected to decrease because of the context-switching penalties. We assess the performance impact by comparing the victim's execution time with and without spy running. To be noticed is that we launch slow-down attacks with more spy kernels, further dampening the victim's performance.

In our slow-down attack settings, there are 8 kernels used by the spy program and it takes 20.9 seconds for the victim to run one VGG16 iteration. In contrast, when no spy is running, the victim's execution time is 431.18ms, indicating 48.5 times slow down. The number of kernels employed by the spy can be used to adjust the ratio of slow-down. Specifically, with only one kernel, the victim's execution time is 637.78ms. As described in Section IV, as training takes hours or days, the slow-down attack is not easily noticeable.

VI. DISCUSSION

Limitations. 1) We evaluated `MoSConS` on Nvidia GeForce GTX 1080 TI. Due to the expense and timing constraints, we did not experiment with other GPUs. Also, to avoid legal issues, we did not test `MoSConS` on the public cloud. On the other hand, we believe our attack should be effective on other GPUs and cloud if the same scheduling and context switching mechanisms are used and the performance counters can be read by the spy. 2) Due to spy's low sampling rate, misclassifications are prone to occur when an op takes a short time. We launched slow-down kernels to alleviate this issue but for the ops that are too short (like `ReLu`) or executed at the end of the iteration, the inference accuracy drops. We will investigate how to address this issue as the next step. 3) `MoSConS` can reveal some critical hyper-parameters but not all defined by a model, e.g., shortcut. In addition, the hyper-parameters not seen by the adversary before cannot be recovered. 4) So far, `MoSConS` is designed to infer a model secret from a single GPU. We will expand `MoSConS` to multi-GPU and distributed-learning settings. 5) We tested `MoSConS` with two users sharing the same GPU. When more users are active, the accuracy of `MoSConS` is expected to decrease as the kernel execution becomes more non-deterministic. 6) `MoSConS` is not supposed to be effective on RNN models due to their very different designs. 7) `MoSConS` is effective on victim models of reasonable complexity, like VGG16. It is expected to be effective when the model size grows, e.g.,

TABLE IX
OVERALL RESULT (EXPLANATION OF OTHER LETTERS ARE IN TABLE VII). RED LETTERS ARE MISCLASSIFICATIONS. ORANGE LETTERS ARE ABOUT VERY SHORT OPS (FINISHED WITHIN 0.8MS). $Accuracy_L$ AND $Accuracy_{HP}$ ARE ACCURACY FOR THE LAYERS AND HYPER-PARAMETERS.

Model	**Ground-truth** **Predicted structure**	$Accuracy_L$	$Accuracy_{HP}$
Cust. MLP	$M_{64,R} - M_{512,T} - M_{1024,S} - M_{2048,R} - M_{8192,T} - Optimizer_{GD}$ $M_{64,R} - M_{512,T} - M_{1024,S} - M_{2048,R} - M_{8192,T} - Optimizer_{GD}$	100.0%	100.0%
ZFNet	$C_{7,96,2,R} - P - C_{5,256,2,R} - P - C_{3,512,1,R} - C_{3,1024,1,R} - C_{3,512,1,R} - P - M_{4096,R} - M_{4096,R} - M_{1000,R} - Optimizer_{Adam}$ $C_{7,96,2,R} - P - C_{3,256,2,R} - P - C_{3,96,1,R} - C_{3,1024,1,R} - C_{3,512,1,X} - P - M_{4096,X} - M_{4096,X} - M_{1000,X} - Optimizer_{Adam}$	100.0%	76.9%
VGG16	$C_{3,64,1,R} - C_{3,64,1,R} - P - C_{3,128,1,R} - C_{3,128,1,R} - P - C_{3,256,1,R} - C_{3,256,1,R} - C_{3,256,1,R} - P - C_{3,512,1,R} - C_{3,512,1,R} - C_{3,512,1,R} - P - C_{3,512,1,R} - C_{3,512,1,R} - C_{3,512,1,R} - P - M_{4096,R} - M_{4096,R} - M_{1000,R} - Optimizer_{Adam}$ $C_{3,64,1,R} - C_{3,64,1,R} - P - C_{3,128,1,R} - C_{3,128,1,R} - P - C_{3,256,1,R} - C_{3,64,1,R} - C_{3,256,1,R} - P - C_{3,512,1,R} - C_{3,512,1,R} - C_{3,128,1,R} - X - C_{3,512,1,R} - C_{5,256,1,P} - C_{3,512,1,X} - P - M_{4096,X} - M_{4096,X} - M_{1000,X} - Optimizer_{Adam}$	95.2%	82.8%

VGG19. However, for more complex models like ResNet50 with shortcuts, MoSConS is unlikely to infer the model structure accurately.

Potential defense. To protect the model secret against MoSConS, the intuitive approach is to restrict access to CUPTI. However, as described in Section II-D, even though Nvidia released a patch [47] as mitigation, it can be bypassed. As such, we believe more principled defense mechanisms are needed. Reducing the precision of CUPTI can interfere with the spy, but again it could introduce side-effect to the legitimate applications. Alternatively, GPU can run a daemon process that detects anomalous contention [10]. In addition, the GPU schedulers (e.g., time-sliced scheduler and warp scheduler) could be enhanced to protect the critical GPU applications (e.g., TensorFlow) and reduce the frequency of preemption by other suspicious applications. We will implement and evaluate those potential defense mechanisms.

VII. RELATED WORKS

The research about model stealing with side-channel [5], [13], [23]–[25], [41], [63], [65] has been summarized in Section I. Naghibijouybari et al. [41] is closest to our work on GPU but only the neuron number of the input layer is recovered. For this task, MoSConS can infer the neuron number of every layer. Below we review other related works.

Confidentiality of machine learning. Our research studies how machine-learning confidentiality can be breached from hardware side. Another direction is to look into the weakness of machine-learning *algorithms*. Research showed that the information about data providers [14], [15], properties of the training data [3], [17], membership of a sample [20], [34], [56], model parameters [48], [59], [61] can be inferred when the model developer publishes its model or allows public API access. An interesting future work could be combining the attacks at algorithm and hardware layers.

Information leakage on GPU. GPU is widely used for encryption and graph rendering besides machine learning. Prior works showed that encryption key can be inferred through timing and power side-channels [27], [28], [35], [50]. Websites visited by a user can be inferred as well [32], [41], [68] GPU side-channels have also been exploited for key loggers [30], row-hammer attacks [16] and building cover-channels [40].

CPU port contention. Our attack introduces contention on GPU units for information leakage attacks. A similar contention-based attack has been explored under CPU execution port [2], [6], showing secrets from OpenSSL can be leaked. Our study extends the research of contention-based side-channel by investigating different hardware, i.e., GPU.

VIII. CONCLUSION

In this paper, we systematically analyzed the issue of information leakage when training a DNN model on a shared GPU. We found that the GPU context-switching penalty can be exploited to allow a spy to obtain finer-grained information of another CUDA application, including DNN ops and hyper-parameters. By leveraging active slow-down attack and passive inference based on LSTM models, we are able to achieve good accuracy for those attack tasks. For future work, we will test the potential defenses as mentioned and we also call on the community and stakeholders to come up with new protection mechanisms schemes to mitigate this risk.

ACKNOWLEDGMENT

The authors would like to thank the insightful reviews and suggestions from our shepherd Dr. Chengmo Yang and anonymous reviewers. The authors also thank Hao Chen and Suprith Ramanan from UC Irvine for their help. The researchers from Fudan University are supported by NSFC 61802068 and Shanghai Sailing Program 18YF1402200. This material is based upon work partially supported by the United States Office of Naval Research (ONR) under contract N00014-17-1-2499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research or its Contracting Agents.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

[3] Giuseppe Ateniese, Giovanni Felici, Luigi V Mancini, Angelo Spognardi, Antonio Villani, and Domenico Vitali. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *arXiv preprint arXiv:1306.4447*, 2013.

[4] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V. Krishnamurthy, Thomas F. La Porta, Patrick D. McDaniel, and Lisa M. Marvel. Malicious co-residency on the cloud: Attacks and defense. In *INFOCOM*, pages 1–9. IEEE, 2017.

[5] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 515–532, Santa Clara, CA, August 2019. USENIX Association.

[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 785–800, New York, NY, USA, 2019. ACM.

[7] BigML. Bigml.com. https://bigml.com/, 2019.

[8] Caffe. Brewing imagenet. https://caffe.berkeleyvision.org/gathered/examples/imagenet.html, 2019.

[9] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

[10] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 216–228, Washington, DC, USA, 2014. IEEE Computer Society.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[12] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 561–574. ACM, 2017.

[13] Vasisht Duddu, Debasis Samanta, D. Vijay Rao, and Valentina E. Balas. Stealing neural networks via timing side channels. *CoRR*, abs/1812.11720, 2018.

[14] Matthew Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333. ACM, 2015.

[15] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 17–32, 2014.

[16] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 195–210, 2018.

[17] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–633. ACM, 2018.

[18] Google. Cloud machine learning engine. https://cloud.google.com/ml-engine/, 2019.

[19] Google. Tensorflow timeline. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/client/timeline.py, 2019.

[20] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. Logan: evaluating privacy leakage of generative models using generative adversarial networks. *arXiv preprint arXiv:1705.07663*, 2017.

[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[23] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitras. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *CoRR*, abs/1810.03487, 2018.

[24] Xing Hu, Ling Liang, Lei Deng, Shuangchen Li, Xinfeng Xie, Yu Ji, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Neural network model extraction attacks in edge devices by hearing architectural hints. *CoRR*, abs/1903.03916, 2019.

[25] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[27] Zhen Hang Jiang, Yunsi Fei, and David R. Kaeli. A complete key recovery timing attack on a GPU. In *HPCA*, pages 394–405. IEEE Computer Society, 2016.

[28] Zhen Hang Jiang, Yunsi Fei, and David R. Kaeli. A novel side-channel timing attack on gpus. In *ACM Great Lakes Symposium on VLSI*, pages 167–172. ACM, 2017.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[30] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[31] Nikolay Laptev, Jason Yosinski, Li Erran Li, and Slawek Smyl. Time-series extreme event forecasting with neural networks at uber. In *International Conference on Machine Learning*, volume 34, pages 1–5, 2017.

[32] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 19–33, Washington, DC, USA, 2014. IEEE Computer Society.

[33] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. When good becomes evil: Keystroke inference with smartwatch. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1273–1285, New York, NY, USA, 2015. ACM.

[34] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *CoRR*, abs/1802.04889, 2018.

[35] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David R. Kaeli. Side-channel power analysis of a GPU AES implementation. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, pages 281–288, 2015.

[36] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. In *Proceedings*, page 89. Presses universitaires de Louvain, 2015.

[37] Microsoft. Lightgbm. https://github.com/microsoft/LightGBM, 2019.

[38] Thorben Moos, Amir Moradi, and Bastian Richter. Static power side-channel analysis-a survey on measurement factors. *IACR Cryptology ePrint Archive*, 2018:676, 2018.

[39] MXNet. A scalable deep learning framework. https://mxnet.apache.org/, 2019.

[40] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael B. Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 354–366, 2017.

[41] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2139–2153, New York, NY, USA, 2018. ACM.

[42] Nvidia. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[43] Nvidia. Multi-process service - nvidia developer documentation. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2018.

[44] Nvidia. Cuda zone. https://developer.nvidia.com/cuda-zone, 2019.

[45] Nvidia. Cupti cuda toolkit documentation. https://docs.nvidia.com/cuda/cupti/index.html, 2019.

[46] Nvidia. Pascal gpu architecture. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/, 2019.

[47] Nvidia. Security bulletin: Nvidia gpu display driver - february 2019. https://nvidia.custhelp.com/app/answers/detail/a_id/4772, 2019.

[48] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. *International Conference on Learning Representations*, 2018.

[49] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414. IEEE, 2018.

[50] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA leaks: a detailed hack for CUDA and a (partial) fix. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):15, 2016.

[51] PyTorch. Cuda semantics pytorch master documentation. https://pytorch.org/docs/0.3.1/notes/cuda.html\#cuda-streams, 2017.

[52] PyTorch. Pytorch. https://pytorch.org/, 2019.

[53] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[54] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[55] Pulkit Sharma. 5 amazing deep learning frameworks every data scientist must know (with illustrated infographic). https://www.analyticsvidhya.com/blog/2019/03/deep-learning-frameworks-comparison/, 2019.

[56] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy*, pages 3–18. IEEE Computer Society, 2017.

[57] Karen. Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[58] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[59] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016.

[60] VMware. Performance and use cases of vmware directpath i/o for networking. https://blogs.vmware.com/performance/2010/12/performance-and-use-cases-of-vmware-directpath-io-for-networking.html, 2010.

[61] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 36–52, 2018.

[62] Chen Wang, Xiaonan Guo, Yan Wang, Yingying Chen, and Bo Liu. Friend or foe?: Your wearable devices reveal your personal pin. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 189–200, New York, NY, USA, 2016. ACM.

[63] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang 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*, ACSAC '18, pages 393–406, New York, NY, USA, 2018. ACM.

[64] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[65] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[66] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[67] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Home-Alone: Co-residency detection in the cloud via side-channel analysis. In *32nd IEEE Symposium on Security and Privacy*, pages 313–328. IEEE Computer Society, 2011.

[68] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable GPU memory management: Towards recovering raw data from GPU. *PoPETs*, 2017(2):57–73, 2017.