IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 587–613. DOI:10.46586/tches.v2023.i2.587-613

"Whispering MLaaS"

Exploiting Timing Channels to Compromise User Privacy in Deep Neural Networks

Shubhi Shukla[1], Manaar Alam[2], Sarani Bhattacharya[3], Pabitra Mitra[1] and Debdeep Mukhopadhyay[1]

[1] Indian Institute of Technology Kharagpur, India,
shubhishukla@kgpian.iitkgp.ac.in,{pabitra,debdeep}@cse.iitkgp.ac.in
[2] New York University Abu Dhabi, UAE, alam.manaar@nyu.edu
[3] IMEC Leuven, Belgium, sarani.bhattacharya@imec.be

Abstract. While recent advancements of Deep Learning (DL) in solving complex real-world tasks have spurred their popularity, the usage of privacy-rich data for their training in varied applications has made them an overly-exposed threat surface for privacy violations. Moreover, the rapid adoption of cloud-based *Machine-Learning-as-a-Service* (MLaaS) has broadened the threat surface to various remote *side-channel attacks*. In this paper, for the first time, we show one such privacy violation by observing a data-dependent timing side-channel (naming this to be *Class-Leakage*) originating from *non-constant time* branching operation in a widely popular DL framework, namely *PyTorch*. We further escalate this timing variability to a practical inference-time attack where an adversary with *user level privileges* and having *hard-label black-box access* to an MLaaS can exploit Class-Leakage to compromise the privacy of MLaaS users. DL models have also been shown to be vulnerable to *Membership Inference Attack* (MIA), where the primary objective of an adversary is to deduce whether any particular data has been used while training the model. *Differential Privacy* (DP) has been proposed in recent literature as a popular countermeasure against MIA, where inclusivity and exclusivity of a data-point in a dataset cannot be ascertained by definition. In this paper, we also demonstrate that the existence of a data-point within the training dataset of a DL model secured with DP can still be distinguished using the identified timing side-channel. In addition, we propose an efficient countermeasure to the problem by introducing *constant-time* branching operation that alleviates the Class-Leakage. We validate the approach using five pre-trained DL models trained on two standard benchmarking image classification datasets, *CIFAR-10* and *CIFAR-100*, over two different computing environments having *Intel Xeon* and *Intel i7* processors.

Keywords: PyTorch Vulnerability · Timing Side-Channel · Differential Privacy

1 Introduction

In recent years, we have seen a growing importance of applications requiring assistance of Deep Learning (DL) in both industry and academia for its undeniable performances in long-standing AI tasks in domains such as image recognition [HZRS16], machine translation [BCB15], malware detection [VAS+19], and autonomous driving [GTCM20] and more. Crowd-sourcing technology giants like Google, Facebook, Amazon, and others collect massive amounts of training data from their users and deploy personalized DL applications on a large scale. The complexity, dynamism, and volume of data in the real world have recently promoted cloud-based Machine Learning as a Service (MLaaS) [RGC15],

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

which provides infrastructural support of powerful computing resources and substantial domain expertise to train efficient DL models. MLaaS is even getting popular among healthcare software solutions that operate on private medical datasets [Ray17, Mej19]. The growing popularity and easy availability of MLaaS has increased the concerns towards protecting user privacy.

There is a relevant field of study pertinent to privacy-preserving DL algorithms. These algorithms use computationally intensive secure multi-party computation [KVH+21, KPPS21] and homomorphic encryption [GDL+16, LJ19] to algorithmically protect user data from entities other than the user itself. **However, in this paper, we illustrate the privacy issues in an MLaaS framework from the perspective of a passive adversary.** We consider that a trusted entity trains a private dataset and designs an implementation for the trained model. We assume that a passive adversary has *hard-label black-box API access*[1] to the trained model and *can obtain information by exploiting the weakness in the implementation of such models* (popularly known as side-channel information). We attempt to address the security issues in this scenario, which is practical and different from the context of the aforementioned field of study.

Related Works

A considerable number of research utilizes the side-channel information in remote environments to reverse-engineer architecture and parameters of Deep Neural Networks (DNN) that are proprietary. Cache-based side-channels have been exploited to reconstruct crucial architectural secret of the victim DNN during the inference phase using the Generalized Matrix Multiply (GEMM) of the victim DNN implementation [HDK+18, YFT20]. Shared GPU resources coupled with hardware performance counters and GPU context-switch side-channel have also been exploited to extract victim's internal DNN architecture [NNQA18, WZZ+20]. It is also shown that by exploiting rowhammer fault injection on DRAM modules crucial parameters of victim DNN can be stolen [RCYF21]. Patterns in memory accesses have also been exploited to steal model weights as well [HZS18].Timing side-channel have been exploited to construct an optimal substitute architecture of the victim as well as extract DL models on high performance edge deep learning processing unit [DSRB18, BBJP19, WCJ+21]

Duddu et al. [DSRB18] exploited timing channel for reverse-engineering a commercialized DL model and *Nakai et al.* [NSF21] exploited timing channel in an embedded platform for crafting *adversarial examples*[2]. *Alam et al.* [AM19] and *Wang et al.* [WHP+] also demonstrated that DNNs are vulnerable to leaking label information of an input instance using cache-based side-channel. However, the method proposed by *Alam et al.* requires *super-user privilege* to access *hardware performance counters* of the system executing DL implementations. On the other hand, the method proposed by *Wang et al.* requires full access to the parameters of DL models (i.e., *white-box* access) or needs to acquire the parameters using previous research on reverse engineering [HDK+18, YFT20, NNQA18, WZZ+20, RCYF21, DSRB18]. Having super-user privilege or the white-box access to DL models may not be practical in various applications where security and privacy of users are of utmost importance. **In this paper, we assume that the adversary has hard-label black-box access to DL models from user-space and can compromise privacy without reverse-engineering any DNN parameters.** Table 1 summarises contribution of this paper compared to related works using remote side-channels on DL.

[1] In hard-label black-box access, a client can only obtain actual labels and does not get any knowledge of probabilities associated with predicted labels.

[2] A threat that adds visually imperceptible perturbations to an input of a DNN to cause misclassification.

Table 1: Summary of our contribution compared to related works using side-channels on DL

Paper	Side-Channel	Adversary Capability	Objective
[HDK+18], [YFT20], [AM19]§ [WHP+]	Cache	●, ●, ●, ○	□, □, ■, ■
[NNQA18], [WZZ+20]	GPU	●, ●	□, □
[RCYF21], [HZS18]	Memory	⊘, ●	□, □
[WLL+18], [MTH+21], [XCC+20]	Power	⊘, ⊘, ●	■, ■, □
[BBJP19], [YMY+20], [YKSF19]	Electromagnetic	●, ●, ⊘	□, □, □
[BBJP19], [DSRB18], [WCJ+21], [NSF21]‡ **This Work**	Time	●, ●, ●, ●, ●	□, □, □, ⊠, ■

§ Requires super-user privilege. ‡ Requires physical access.
○ :Full knowledge of DNN. ⊘ :Only DNN architecture known. ● :DNN Black-box access.
□ :Model reverse-engineering. ⊠ :Craft adversarial examples. ■ : Leak label/training dataset information.

Our Contributions

In this paper, we define the term *Class-Leakage* as side-channel-based information leakage from *DL implementations* that aids an adversary in distinguishing unknown labels of different inputs without explicitly accessing the model parameters or the input data. Information on the label of input data is directly linked to the sensitive information of a user, highlighting a critical privacy concern in DL implementations. On the other hand, the continuous rise of DL has propelled the growth of various open-source libraries, like Tensorflow [A+16a], Keras [C+15], PyTorch [P+19], Theano [A+16b], etc., for efficient data flow while implementing DL models. **In this paper, we primarily focus on PyTorch and identified an implementation vulnerability typically responsible for Class-Leakage through timing side-channel[3].** We demonstrate that operation of the *Max Pooling* module in a Convolutional Neural Network (CNN) using PyTorch is vulnerable to input-dependent timing side-channel leakage due to improper *non-constant time* implementation of branching instruction[4]. Further, we also demonstrate that the vulnerability can be exploited during the inference phase of a DL model by an adversary having access to a *manifest dataset* to compromise the privacy of a user using a Multi-Layer Perceptron (MLP). The manifest dataset being the set of *annotated data* that is apparent to the adversary, though not necessarily a subset of the original training dataset but resembling it sufficiently.

In addition to the aforementioned threat to user privacy, the PyTorch vulnerability also exposes a possible threat of *Membership Inference Attack* (MIA) [SSSS17, CTCP21, TLG+21], but this work does not include it. The objective of an MIA adversary is to deduce whether an unknown data has been part of the training dataset of a DL model. Significant efforts in recent literature to prevent MIA – *Differential Privacy* (DP) being a popular choice [ACG+16, NSH20, PTS+21]. DP algorithmically ensures that the output of a DL model does not leak any information whether an unknown data has been a '*member*' of the training dataset or is a '*non-member*'. **In this paper, we demonstrate that the Class-Leakage coupled with an MLP classifier can successfully determine the presence of an unknown data in the training dataset of a DL model protected with DP, countering its privacy guarantees.** We used the PyTorch-based open-source library *Opacus* [YSS+21] to train DL models with DP and demonstrate its vulnerability against side-channel-based Class-Leakage mentioned above.

In this work, we also thrive on implementing an inexpensive countermeasure by making minimal changes to the existing PyTorch library to mitigate the correlation of timing side-channel with input data without affecting the accuracy of DL models. We propose a constant time implementation of Max Pool operation to the existing PyTorch library that does not affect the accuracy of DL models.

[3]The simplicity, ease of use, dynamic computational graph, and efficient memory usage have recently made PyTorch one of the most sought-after libraries for several organizations to implement industrially standard DL applications [Bal20, Car20]. These applications are exposed to the discussed vulnerability.

[4]**We reported the vulnerability to the Meta (Facebook) AI Research team (developer of PyTorch), and was acknowledged. We discuss the vulnerability disclosure in Section 8.**

We demonstrate that the proposed modified constant-time Max Pooling module can successfully mitigate the input-dependent timing side-channel leakage existing in the current PyTorch-based implementation. We also demonstrate that the proposed constant-time Max Pooling module successfully alleviates the vulnerability of differential-private DL models based on PyTorch (like Opacus) against our proposed attack even if the adversary has additional information through timing side-channel.

Real Life Impact

PyTorch's user base has exponentially increased since its release in 2016. Hence, the timing leakage vulnerability observed in the PyTorch library is a serious privacy concern for the parties using it to work with highly confidential data, and it should not be ignored. Major technology giants have built their deep learning models on PyTorch including Microsoft, Tesla, Uber, Airbnb, and Facebook itself. Tesla has built its Autopilot [DB17] system on PyTorch. The adversary could exploit the timing leakage to get the classification result of the models running to predict the car's next move. The predictions can be put together to get the complete route taken by the car. This is a potential scenario of serious privacy threat for the car's owner as well as Tesla. Another possible threat scenario is for healthcare organizations that use cloud services to deploy their deep learning models, for various purposes [Mej19]. The models mainly use datasets containing information from patient health records, with patients' identities kept anonymous for privacy reasons. Using the attack discussed later in Section 5.3 an adversary from some rival company can steal information about the training set data used by the victim healthcare company, by bypassing differential-private models and breaching patients' privacy.

We summarize our **primary contributions** through this paper as follows:

- We identify an *implementation vulnerability in PyTorch's Max Pool operation* leading to *Class-Leakage*, aiding an adversary to distinguish unknown labels of different inputs.
- We demonstrate a methodology where an adversary can exploit the *Class-Leakage* vulnerability during the *inference phase* of a DL model to *compromise the privacy of victims by predicting unknown labels of their inputs* that are directly linked to their sensitive information using an MLP classifier.
- We demonstrate that a DL model secured using differential privacy can *still be vulnerable against an attack to get information about the training data, if the adversary has additional information through timing side-channel-based Class-Leakage.*
- We propose an *easy-to-implement countermeasure* to develop a *constant-time Max Pooling operation* by making minimal changes to the existing PyTorch library that does not affect the accuracy of DL models.
- We evaluated all experiments on standard image classification benchmarking datasets like *CIFAR-10* and *CIFAR-100* [KH+09] using CNN models like *AlexNet* [KSH12], *DenseNet121* [HLvdMW17], *SqueezeNet* [IMA+16], *ResNet50* [HZRS16], and *VGG19* [SZ15]. To validate the generalizability of the method in different computing environments, we performed all the experiments on both Intel Xeon and Intel i7 processors.

The rest of the paper is organised as follows: Section 2 presents a brief overview of the necessary background. Section 3 introduces the data-dependent timing side-channel leakage identified in PyTorch. Section 4 demonstrates a practical inference-time attack using the timing side-channel leakage. Section 5 illustrates the vulnerability of differential-private models using the timing side-channel leakage. Section 6 discusses a proposed countermeasure to alleviate the timing side-channel leakage. Section 7 extends the threat model proposed in Section 4 to mount a more practical end-to-end attack. Section 8 presents a brief discussion on disclosure of the vulnerability to Meta (Facebook) AI research. Finally, Section 9 concludes the paper.

2 Preliminaries on Convolution Neural Networks

Convolutional Neural Networks (CNNs) [AMAZ17] is a type of neural network specialized in processing multidimensional data like images. CNNs derive their unique advantage from their method of processing the data in a grid-like structure to extract useful features. A basic CNN architecture has three main layers: convolution layer, pooling layer, and fully-connected layer. We next provide a brief illustration of the Max Pooling layer, which is pivotal in this paper.

Max Pooling: Pooling layers [GK20] provide a method for down-sampling feature maps by summarizing the existence of features in patches of feature maps. There are two types of pooling functions – Max Pool and Average Pool. The pooling function is defined by kernel size, stride, and padding. *Kernel size* is the window size of the sub-matrix on which pooling operation is applied on the feature map. In Max Pool, the maximum value among all elements in the window is the output for that window. *Stride* defines the step size of the window both in the vertical and horizontal direction. *Padding* determines the number of extra rows and columns with zero appended to the output matrix to regulate its size. Figure 1 shows an example of Max Pooling operation with stride 2, kernel size of 2×2, and zero padding.

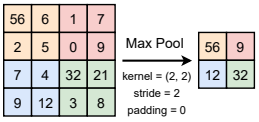

Figure 1: Example of Max Pooling operation

3 Timing Leakage Vulnerability in PyTorch

The Python-based Deep Learning (DL) library PyTorch, developed by Meta (Facebook) AI Research, has recently found a growing interest in several industry-standard AI-enabled products because of its dynamic graph creation, data parallelism debugging, and developer-friendliness. The popularity of PyTorch even pushed several organizations to replace famous DL stacks like TensorFlow with PyTorch as a core module in various applications [Bal20, Car20]. Even though PyTorch is a prevalent and powerful library, we have identified a data-dependent vulnerability that can leak class-label information of inputs through the timing side-channel. The identified timing side-channel can lead to serious privacy violations to the user base of a Machine Learning as a Service (MLaaS) provider, which we have discussed later in Section 4. In this section, we first show the timing leakage vulnerability in PyTorch-based Convolution Neural Networks (CNN). Then we identify, demonstrate and analyze the source of timing leakage. In the following subsection, we provide details on the experimental scenario and basic setup we used for our analysis performed in this section.

3.1 Experimental Scenario and Setup

Scenario: We consider a remote cloud server that provides MLaaS to its clients, and the cloud server uses PyTorch to implement its CNN model. We assume that a client has hard-label black-box access to CNN, i.e., the client can only query CNN with input and obtain classification output as a hard-label. In hard-label black-box access, a client can only obtain actual labels and does not get any knowledge of probabilities associated with predicted labels. We also assume that the client can monitor execution times during the inference operation with user-level privilege. The scenario is briefly illustrated in Figure 2.

Setup: In order to first establish the timing side-channel leakage, we consider a custom CNN architecture implemented using PyTorch (1.9.1+cpu). The architecture of custom CNN is provided in Table 2. We consider two widely-used standard image classification benchmarking datasets, *CIFAR10* and *CIFAR100*, for the evaluation. We also analyze

several pre-trained CNNs implemented using PyTorch, namely *AlexNet*, *DenseNet121*, *SqueezeNet*, *ResNet50*, and *VGG19*, to support the claim of timing side-channel leakage. In order to validate the generalizability of timing side-channel leakage in multiple computing environments, we perform all our experiments on an *Intel Xeon* (4 cores, Skylake) machine with 16GB RAM and an *Intel i7-4790* (4 cores, Haswell) machine with 16GB RAM. In the following subsection, we provide an overview of the timing measurement strategy and details on analyzing the timing.

3.2 Analysis of Timing Measurements

We use the `perf_counter()` [TIM] method from Python's `time` library to obtain execution time of a CNN during its inference operation for a particular input. The `perf_counter()` method can be invoked using user-level privilege. The code snippet in Listing 1 (Appendix A inside supplementary material) is a sample example of obtaining the total execution time of a *forward propagation* (i.e., inference operation of PyTorch library) for a sample CNN model. The term $t2 - t1$ provides the overall inference time for input x.

Let the inference time for any input k of class $\mathcal{C}_i$ be denoted as $t_{\mathcal{C}_{i,k}}$. We observe $t_{\mathcal{C}_{i,k}}$ for $\mathcal{N}$ repetitions to obtain a distribution $\mathcal{T}_{\mathcal{C}_{i,k}} = \{t^1_{\mathcal{C}_{i,k}}, t^2_{\mathcal{C}_{i,k}}, \ldots, t^{\mathcal{N}}_{\mathcal{C}_{i,k}}\}$, where $t^r_{\mathcal{C}_{i,k}}$ is the value of $t_{\mathcal{C}_{i,k}}$ at r-th repetition. We repeat the process for $\mathcal{P}$ different images for a particular class $\mathcal{C}_i$ to make the analysis more generalised over different input instances and thus obtain the *timing distribution* $\mathcal{T}_{\mathcal{C}_i} = \{\mathcal{T}_{\mathcal{C}_{i1}} \parallel \mathcal{T}_{\mathcal{C}_{i2}} \parallel \ldots \parallel \mathcal{T}_{\mathcal{C}_{i\mathcal{P}}}\}$, where $\parallel$ is the append operation and $|\mathcal{T}_{\mathcal{C}_i}| = \mathcal{PN}$. Let the number of classes in our analysis be $\mathcal{Z}$. We thus have a total of $\mathcal{Z}$ timing distributions. We take two timing distributions at a time and perform t-Test on all pair of distributions (i.e., for $\binom{\mathcal{Z}}{2}$ pairs). For a pair of distributions $(\mathcal{T}_{\mathcal{C}_i}, \mathcal{T}_{\mathcal{C}_j})$, we calculate the t-statistic [KHL11]. We report that a given input of class $\mathcal{C}_i$ is distinguishable from a given input of class $\mathcal{C}_j$ if $|t| > 4.5$ with a confidence of 0.99999.

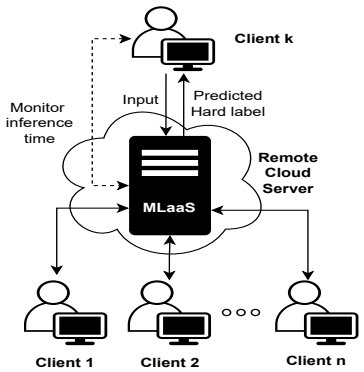

Figure 2: Experimental Scenario: A client having hard-label black-box access to a remote cloud-server providing MLaaS through a CNN implemented on PyTorch. The client can monitor execution time during inference operation

Layer Name	Layer Type
Input Layer	Input
Layer1	Convolution-16
Layer2	Convolution-32
Layer3	MaxPool
Layer4	Convolution-32
Layer5	Convolution-32
Layer6	MaxPool
Layer7	Convolution-64
Layer8	Convolution-128
Layer9	Fully connected - 128
Layer10	Fully connected - 64
Output Layer	Softmax

Table 2: Custom CNN Architecture for CIFAR10 and CIFAR100. The filter size of Convolution layers and the number of neurons in Fully Connected layers are given after layer names. MaxPool's Kernel size = 3x3 and Stride = 2

3.2.1 Inference Time Analysis

We perform the inference time analysis, as discussed above, for the custom CNN mentioned in Table 2. We observe that out of 45 class pairs in CIFAR-10, 42 class pairs can be distinguished. In CIFAR-100 out of 4950 class pairs, 4222 class pairs can be distinguished based on

the inference time in Intel Xeon machine[5]. In Intel i7 machine, the results for CIFAR-10 and CIFAR-100 are 37 and 3265, respectively. The high number of distinguishable pairs for both datasets indicates data-dependent timing-leakage, generalized over different computing environments.

We perform the same analysis on five different pre-trained CNN models, as discussed in Section 3.1, to investigate the existence of data-dependent timing side-channel leakage in other CNNs. The results of inference time analysis for all these models considering CIFAR-10 and CIFAR-100 datasets for both Intel Xeon and Intel i7 machines are shown in Figure 3a and Figure 3b, respectively. The vertical axis in the figure represents the total number of distinguishable class pairs for CIFAR-10 (red) and CIFAR-100 (blue). We can observe that most class pairs in CIFAR-10 and CIFAR-100 can be distinguished in all these models using only the inference time.

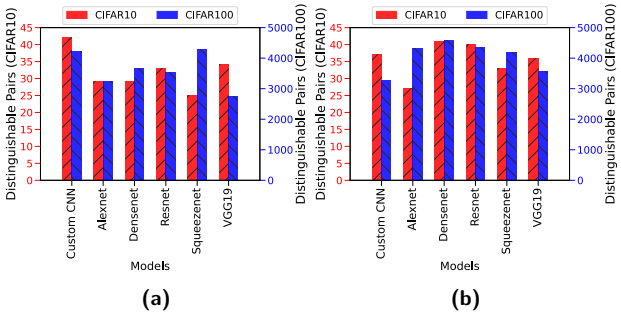

Figure 3: Number of distinguishable class pairs using timing side-channel in different CNNs on CIFAR10 (out of 45) and CIFAR100 (out of 4950) on (a) Intel Xeon and (b) Intel i7

3.2.2 Layer-wise Inference Time Analysis

From the previous discussion we observe that inputs from different class labels can be distinguished from each other based on inference timing. In order to investigate the source behind the existence of this data-dependent timing side-channel leakage, we perform the same analysis using the execution time of each layer during the inference phase. The code snippet in Listing 2 (Appendix A inside supplementary material) is a sample example of obtaining layer-wise execution time during the inference operation of a sample CNN. We accumulate different timestamps into variable t after the execution of each layer during forward propagation.

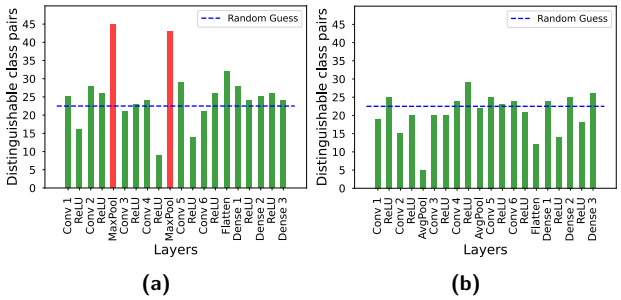

Figure 4: Number of distinguishable class pairs (out of 45) using timing side-channel of each layer in Custom CNN on CIFAR10 with (a) Max Pooling (b) Average Pooling

From the final values in t, we can compute the execution time of each layer by subtracting adjacent values. We perform inference time analysis on custom CNN considering CIFAR-10 on Intel Xeon machine to obtain total number of distinguishable class pairs using execution times of each layer. The result of the analysis is shown in Figure 4a. The vertical axis in the

[5]CIFAR-10 has images of 10 classes indicating a total of $\binom{10}{2} = 45$ class pairs, and CIFAR-100 has images of 100 classes indicating a total of $\binom{100}{2} = 4950$ class pairs.

figure represents the total number of distinguishable class pairs for each layer represented in the horizontal axis[6]. We can observe that all layers can distinguish different numbers of class pairs. In case on no timing leakage the layers will only distinguish approximately around 50% pairs or less than that, which is equivalent to a random guess. *However, the Max Pooling layer (highlighted in red) can distinguish the maximum number of pairs, much greater than a random guess. Hence, it is considered to be the primary contributor to observed timing differences using the overall inference time.*

3.2.3 Timing Analysis with Average Pool

Average Pooling[7] is another very commonly used pooling function in CNN architectures, hence we try to verify whether this vulnerability is limited to max pooling or found in other pooling functions as well. We repeat the layer-wise timing analysis by replacing all MaxPools in the Custom CNN model with Average Pool. The results are given in Figure 4b. *We observed that whereas the max-pooling layers could completely distinguish all class pairs, the average pooling layer was only able to distinguish less than fifty percent of the class pairs, which is equivalent to random selection.* Hence, we experimentally conclude that Average Pool does not leak data-dependent timing information. We further explore the cause behind the leakage in MaxPool in the next subsection.

3.3 Analysis on PyTorch Maxpool Implementation

The MaxPool function slides through defined kernel size matrices to get reduced feature maps as shown in Figure 1. Ideally, the implementation of MaxPool should take constant-time for all inputs. For the sake of performance optimization, libraries are usually designed without much consideration given to constant-time implementation of functions, which can be exploited using side-channels. In Figure 4a we observed that MaxPool can distinguish between almost all class pairs, signifying the lack of constant-time implementation. Hence, we delve into the PyTorch implementation of Maxpool function to analyze the cause of the data-dependent timing leakage. The code snippet from PyTorch Github repository[8] is shown in Listing 1.

```
1 if ((val > maxval) || std::isnan(val)) {
2     maxval = val;
3     maxindex = index;
4 }
```

Listing 1: PyTorch Maxpool's code snippet to update max

The implementation uses an 'if' statement to get the maximum value for each pooling window, which checks all elements and keeps updating the max value when it finds a greater value. It also updates the index (*maxindex*) of the current max value. In Listing 1, *val* is the value of the element at current *index* and *maxval* stores the value of the maximum element found till now in the current window. For each window, the number of times assignment operation inside if statement is executed depends on the position of the max value in the window. Hence, the overall number of assignment operations executed inside the 'if' statement differs for different inputs to Maxpool. This is illustrated in Figure 5 using an example. The yellow 3×3 windows represent the current window of the input matrix on which the Maxpool is being operated. The bold green text emphasizes on the indices of the matrix, for which the assignment operation inside the 'if' statement has been executed. We see in the figure, that for two different inputs the assignment operation inside the 'if' statement vary in both kernel-sized windows of both the inputs. *The total*

[6] We consider activation functions as separate layers.

[7] Average Pooling reduces feature map dimension by averaging the values instead of finding maximum.

[8] PyTorch Github repository code snippet (line 65-68): https://github.com/pytorch/pytorch/blob/bceb1db885cafa87fe8d037d8f22ae9649a1bba0/aten/src/ATen/native/cpu/MaxPoolKernel.cpp#L65

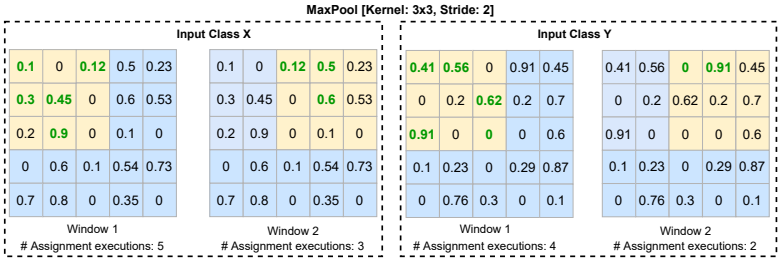

Figure 5: An example of varying number of if statement calls for two different windows (Kernels) during Maxpool operation

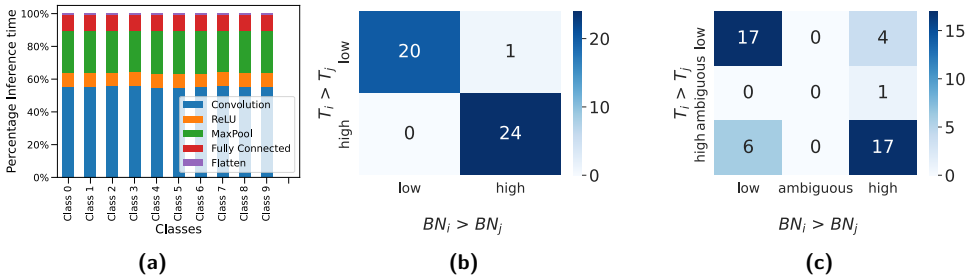

Figure 6: (a) Percentage of time taken by each layer type in single inference for all 10 classes (b)First Maxpool Layer Timing correlation with Branch not taken instructions (c) Second Maxpool Layer Timing correlation with Branch not taken instructions

assignment executions for input X and Y differ from each other, hence causing difference in the execution time of the MaxPool operation and the overall inference time of an input.

Since the number of times the assignment operations inside 'if' (branch not taken) statement is executed depends on the input to the MaxPool function, it makes the implementation dependent on the input data given to the MaxPool function. The input data is resultant of multiple transformations (like convolution and activation operations) applied to the original input provided to the DL model, hence there is a correlation between the two, which indirectly makes the total number of assignment operations dependent on the input to the model. Average Pool [9] does not contain such 'if' statement and thus shows resilience to information leakage, which validates the results in Figure 4b.

3.3.1 Influence on Timing

In this work we show how an adversary can exploit this implementation vulnerability to know the class labels of the input data using timing side-channels. It is possible because, different class labels execute different number of 'if' statements inside the MaxPool implementation. This causes a timing variation in the overall inference time for all inputs hence aiding an adversary to distinguish among them the different class labels. It has a significant impact on timing because almost 25% of the total inference time is taken by the pooling functions (green) in our custom CNN model as shown in Figure 6a. From the figure we can see that convolution operation (blue) takes the maximum time in the inference (more than 50%) and the pooling is the second most time consuming operation. We verify this with experimental analysis on the custom CNN model to do the analysis which has two Maxpool layers. We get input tensors to both MaxPool functions for one

[9] PyTorch Github repository Average Pool code snippet https://github.com/pytorch/pytorch/blob/bceb1db885cafa87fe8d037d8f22ae9649a1bba0/aten/src/ATen/native/cpu/AvgPoolKernel.cpp#L15

image of each class. We calculated the number of 'if' statement executions (branch not taken instructions) for all class images for both max pools. We compared these values for all class pairs and observed that the results were directly proportional to the timing results i.e., when the number of 'if' statement executions for an input class is higher than another class, its execution time will also be higher. The results are shown in Figure 6(b) and 6(c). Here T_i and T_j are median of inference time distributions of some class pair C_i and C_j and BN_i is the number of branch not taken (if statements execution) instructions executed by max pool function for class C_i. There are 45(10C2) pairs total, hence total of 45 cases in the confusion matrices shown in Figure 6(b) and 6(c). *High* indicates when $T_i > T_j$ and $BN_i > BN_j$, whereas *Low* indicates vice-versa. For instance, the first confusion matrix with 4 tiles in Figure 6(b), represents 4 cases. Top left tile indicates $T_i < T_j$ and $BN_i < BN_j$ 20 out of 45 times, bottom right tile indicates that the other 24 times, $T_i > T_j$ and $BN_i > BN_j$. Top right tile with value 1 means, $T_i < T_j$ and $BN_i > BN_j$, and no cases with $T_i > T_j$ and $BN_i > BN_j$ both are aberrations. In Figure 6(c), *Ambiguous* indicates the cases where we are not able to distinguish a pair based on timing or branches, because they are almost similar for both classes of the pair. From the confusion matrices in Figure 6 we can observe that, in majority of the cases the inference time T_i of a class label input i is greater than inference time T_j of a class label input j when branch not taken instructions of the former is greater than the later and vice-versa. In the next section, we discuss on a practical threat model where an adversary can exploit the class-leakage to compromise user privacy.

4 Timing based MLP Class-Label Classifier

In the previous section, we identified the primary component responsible for timing leakage in the PyTorch library, and the effect of such leakages is visible over several popular datasets. But the observed non-constant timing behavior can lead to a practical end-to-end attack. In the following discussion, we first define the threat model and then demonstrate an attack to infer class labels of unknown inputs using the timing leakage.

4.1 Threat Model

Let us consider a scenario where multiple clients are connected to a trusted cloud server providing Machine Learning as a Service (MLaaS), as demonstrated in Figure 7a. Clients provide their private/confidential data to the MLaaS to get back classification results for a particular task. The clients have no more than user-level privileges, but they could be curious and possibly have malicious intention of knowing the private data of another client. Given this scenario we present the capabilities and objective of the adversary as follows.

Adversary's Capability and Objective: The adversary is considered to have a hard-label black-box access to the DL model on the MLaaS server with user-level privilege. The adversary also has an access to a manifest dataset. Manifest dataset is a set of inputs which may or may not be in the training dataset but belongs to the same distribution. We consider this to be a realistic assumption because attacker and victim are both clients of same MLaaS model, so apparently, they participate in inferences of resembling input classes, signifying their distributions are congruent. This is a commonly accepted notion in MLaaS [NSH19, PMG+17]. *It is to be noted that, the adversary and the victim client connect to the MLaaS server remotely over the network and have access to the same deep learning model.* The adversary does not have the ability to get timestamps of start and end of image inference from inside the victim's code. With these capabilities, adversary's objective is to infer class labels of input data fed by the chosen victim client to the MLaaS. In addition, the adversary is also capable of launching a profiling attack using the timing channel leakages. The adversary connects to MLaaS over an *ssh* connection and has no

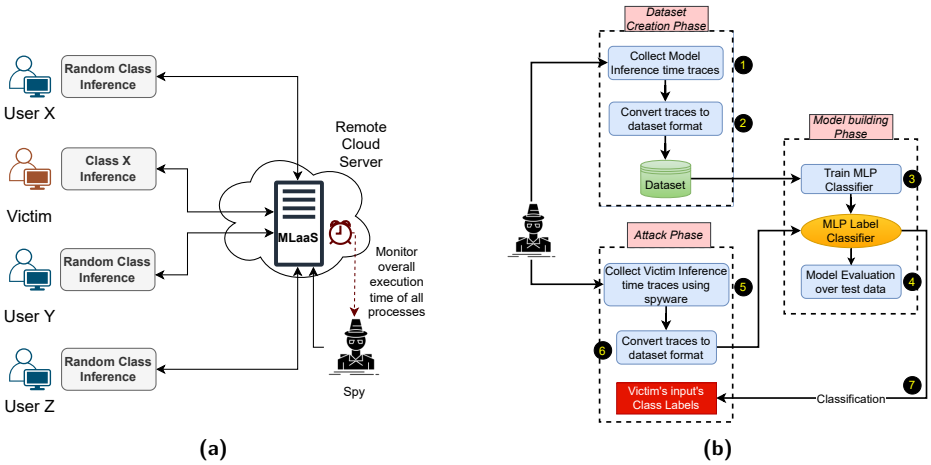

Figure 7: (a) Threat Model for Privacy Leakage in a Client-Server MLaaS Framework (b) Attack Methodology: The process goes through three stages (i) Dataset creation Phase, (ii) Model Building Phase and (iii) Attack Phase (The numbered circles indicate the order of the process)

physical access to the system on which the ML model is deployed.

4.2 Timing Analysis in presence of Multiple Background Processes

We have demonstrated the timing leakage in a noiseless setting so far in our previous discussion. We assumed that only one process executes in the system. However, in reality, there can be multiple processes accessing the MLaaS at any point of time. Hence, the parallel processes running in the background may also add execution noise to the inference time of the victim process due to sharing of resources. Therefore, before moving on to the profiling attack, we first verify the presence of the PyTorch timing vulnerability in the multiple-process scenario where multiple clients access the MLaaS at the same time and the spy tries to infer victim's inference time *without any communication among the two*. Since the adversary does not have access to timestamps for the starting and ending of the inference operation from inside the victim's code. Hence, the adversary requires a mechanism to detect the start and end of the execution of the victim process. For this purpose, the adversary can launch a script which constantly sniffs the executing processes on the MLaaS server. In particular, this can be done by running a continuous script on the server which monitors the processes running on the server, and as soon as the execution of the victim process is detected the script starts a timer and stops it only when the execution of victim process stops. In a Linux environment, the adversary can monitor the processes with the help of the `ps` command which requires only user-level privileges. The adversary can monitor victim execution using the PID of the victim process.

Experimental Setup: We consider n processes running on a single logical core for our experiments. These processes are run in parallel using the *fork* system call. One of these processes is the victim process, which runs inference function with a batch of images from any particular class of the CIFAR-10 dataset. The spy's objective is to know the class-label of the image batch run by the victim. The other $n-1$ processes run the inference function as well for a batch of inputs, and the batch consists of random images from all classes. The other processes can be considered as other users of the MLaaS server. The victim process runs inference for any class input multiple times. In our experiments we take the number of continuous inferences to be 1000. This is a reasonable assumption considering

batch-inferences [Pat19] scenarios, for instance in healthcare industry and pharmaceutical manufacturers where data privacy is crucial. In such applications, the client usually feeds a batch of inputs from the same class for classification on the MLaaS. For instance, a Covid-19 facility with multiple patients would daily feed the data of all patients to the MLaaS to get the current Covid positive/negative status of each patient at once. It is also not necessary for the victim to run the same image of a class while batch processing. He can also run different images of the same class as well. We discuss this with more practical experiments in Section 7. In the following discussions we consider the victim to run multiple inference of the same input as a proof of concept to demonstrate the existence of timing leakage in a noisy scenario. Further, another possibility is that the adversary has the capability to mute responses on TCP from server which forces the client to re-infer the same input multiple times. By default, the re-transmission timer is set to 240 seconds (4 minutes), which can be modified to a lower value by the adversary, hence enabling back-to-back re-transmissions of the same input.

Retrieving Total Execution Time: In our setup, we try to simulate a parallel environment by using the fork() functionality to execute n processes simultaneously, similar to the previously discussed threat model. The time window between the entry and exit of a victim process is monitored by the public PID information, during which a timestamp counter is run to estimate the victim's inference time. It may be noted, that the estimated time is polluted by the execution times of the other $n-1$ random processes. However, we show that the victim's inference time is exposed to the adversary because of the randomness of the inferences performed by the other parallel processes whose times converge to the means of the respective distributions.

Timing Analysis: We perform the same timing analysis as in Section 3.2 with the custom CNN model by taking $n = 4$ and getting the timing values from the spy process. We observe that 38 (84.4%) out of 45 class pairs are distinguishable in the noisy environment. We confirm the viability of our results showing the vulnerability in the PyTorch library, by proposing a profiling timing attack using a MLP classifier to mount an end-to-end attack.

4.3 MLP Class-label Classifier for End-to-End Attack

The objective of the attack is to analyze whether the MLP is able to learn the timing difference among the different classes. We divide the process in three main parts: Dataset creation, tuning and training the model, and testing the model on completely new data. The complete process flow is performed using the manifest dataset and shown in Figure 7b.

Dataset Creation: Let the inference time for a particular input image k of class C_i , selected from the manifest dataset be denoted as $t_{C_{i,k}}$. We observe inference time $t_{C_{i,k}}$ for N time instances to obtain a timing distribution $\mathcal{T}_{C_{i,k}} = \{t^1_{C_{i,k}}, t^2_{C_{i,k}}, \ldots, t^N_{C_{ik}}\}$ of that particular input. Now repeating this procedure over all inputs of the class, we get $\mathcal{T}_{C_{i,k}}$ for P different inputs of class C_i from the manifest dataset, to get $\mathcal{T}_{C_i} = \{\mathcal{T}_{C_{i,1}}, \mathcal{T}_{C_{i,2}}, \ldots, \mathcal{T}_{C_{i,P}}\}$. Next, representative timing points are generated by selecting the statistic value of the respective distributions. Medians of all P distributions in $\mathcal{T}_{C_i}$ are denoted as $\mathcal{M}_{C_i} = \{\tilde{\mathcal{T}}_{C_{i,1}}, \tilde{\mathcal{T}}_{C_{i,2}}, \ldots, \tilde{\mathcal{T}}_{C_{i,P}}\}$. This $\mathcal{M}_{C_i}$ array makes one row of our dataset with class label C_i. It is to be noted, that we do not require to retrain for every different N, since we did one training for the median of a distribution with N observations. To avoid underfitting, we make our dataset bigger by repeating the same experiment for all classes Z times with $P = 100$, $N = 500$ and $Z = 1000$ respectively. With $Z = 1000$, we get total of 10000 rows in our dataset for 10 classes (1000 rows for each class). With $P = 100$, we have total 100 columns in our dataset, resulting in a final dataset of dimension 10000×100.

Model Training: The dataset as constructed with the representative median samples are split into training (80%) and testing data (20%). We use Scikit-learn's `MLPClassifier` [PVG+12] to build our model. Further, to get the best fit for our model, we used Scikit-learn's `GridSearchCV` functionality which takes in a set of different parameters

such as, activation functions, learning rate, and network size, and then returns the set of hyperparameters which fit the model best in terms of accuracy. For our GridSearchCV space we tested with following set of hidden layers: [(50,50,50), (400, 400, 400,50), (300, 300, 300,300), (200, 200, 200)and (350,200,100, 50)] and learning rates: [0.0001, 0.001, 0.01]. We fixed the activation function and optimizer to ReLU and Adam. Additionally, to avoid over-fitting the model we used K-fold validation method where we chose $K = 10$.
Testing: The testing phase is carried out on the 20% test data which was not used while training. We also test our classifier on a completely new dataset of 2000 rows containing 200 data points for each class label. We used the new data as a confirmatory test that the model was not over-fitting on the split test data.

4.4 Results and Analysis

In this section, we give performance results of the MLP model, trained to classify class-labels using model inference time. As discussed previously, we use MLP classifier with 10000 and 2000 (200 data-points for each of 10 classes) data-points for training/testing and to check over-fitting of the model on fresh data respectively. The data-points are built using inference time values from the image classification model for all classes. For the CNN model, we again select the custom CNN model used in Section 3.2. (Refer to Table 2 for the architecture details). We achieved classification accuracy of approximately 99.35% on our unused test data with 2000 inputs, from the class-label MLP classifier. In Figure 8a, we show the confusion matrix for the classification of the test data. We see that classes *0, 3, 4, 5, 6,* and *7* are classified with 100% accuracy, and misclassification rate for remaining classes is less than 2.5%. These results imply that the PyTorch vulnerability can be exploited using the proposed profiling attack for an end-to-end attack.
Results for Multi-process Attack Scenario: We now show the results for the MLP classifier attack when we work with the multi-process scenario. Earlier we saw in the multiple process experimental setup (Section 4.2), that the victim runs 1000 inferences of any class X, and $n - 1 = 3$ processes execute in parallel. The adversary collects the overall execution time of all four processes combined. For the attack we create a dataset of dimension 1000×100, where for each row the 100 columns consist of overall timing for the four processes, calculated with 100 different images of the same class. Out of these 1000 data points, we separate out a test set of 200 data points. Rest everything remains same as the attack in the previous section. The results are shown in Figure 8b. We get an accuracy of 91% for our test data. For classes, 0, 2, 4, and 9 we get 100% classification accuracy. Next, we discuss the timing analysis results and attack scenarios in the case of differential-private models. We also show results for the attack with $n = 8$ processes, in Figure 8c. It has been shown that the performance of deep learning inferences can be significantly sped up in a single machine by spawning multiple worker processes in vastly popular Amazon AWS framework. However, the performance does not improve significantly after spawning more than eight processes. Hence, in our study, we considered eight processes as a representative of a realistic setup [Wan19]. We created a dataset of dimension 1000×80 for the 8-process attack and observe an accuracy of approximately 83% for the test data of 200 data-points, showing that the classes are still distinguishable. We observe a drop in accuracy because of increased noise caused by addition of more processes, hence by increasing the number of inputs the classification can be improved.

5 Class-Leakage in CNN with Differential Privacy

In Section 3.2 we observed one aspect of PyTorch's timing vulnerability, which is its capability to leak information about the class labels. Now, in this section, we explore another possibility of a certain kind of privacy leakage caused by this vulnerability. In

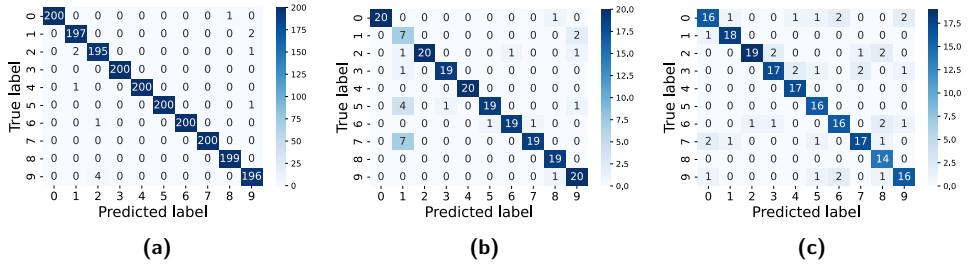

Figure 8: Confusion Matrix for timing test data on the MLP model (Custom CNN) in (a) single process scenario (b) 4-process scenario and, (c) 8-process scenario

the introduction, we briefly discussed about membership inference attack, whose main objective is to identify whether a pair of input and output of a model, belongs to the training set used to train that model. The most common defense against this attack is differential privacy. Differential Privacy adds a certain amount of perturbation to the model hyper-parameters during the training process such that the model does not overfit on the training data. The amount of noise added to the model can be tuned using a parameter called privacy budget denoted by ϵ. The privacy budget is used to balance between model's performance and its privacy protection capability.

It would be interesting to see if we train models with differential privacy, is there a way to leak information about the training data using PyTorch's vulnerability? To begin with, we first verify whether the vulnerability persists after training our model with differential privacy and then see how we can bypass it with the help of the attack proposed earlier.

5.1 Analysis of timing vulnerability with Differential Privacy

In this section, the experimental setup remains the same as Section 3.2 for all experiments. Additionally, we modify the basic Custom CNN model and five other CNN models: Alexnet, Resnet50, Densenet, VGG19, and Squeezenet, by training them with differential privacy. For this purpose, we use *Opacus*[10] [YSS+21] library, a research initiative by Facebook to provide differential privacy to DL models implemented on PyTorch. Next, we begin our analysis by looking into the overall inference time for the differential-private model.

Analysing Overall Inference Time: The experiment structure remains the same from Section 3.2.1, as to analyze the effect of PyTorch vulnerability on our differential-private models. The results are illustrated in Figure 9a. The results confirm that the majority of the class pairs are distinguishable using the timing side-channel. In Table 3 we also compare the percentage of class-pairs distinguishable by the CNN models when they are trained with and without differential privacy. From results in Table 3 we also observe varying distinguishable percentage for all models which is a result of the number of computations from other layers creating noise in timing measurements.

Analysing Layer-wise Inference Time: To further verify that the MaxPool function is the source of leakage for differential-private CNNs as well, we do a layer-wise analysis. Following the experiment steps similar to Section 3.2.2, the results are shown in Figure 9b. On average, the six convolution layers are able to distinguish 18 class-pairs, the eight ReLU activation distinguish 12 class-pairs and the three dense layers distinguish 20 class pairs, which are all less than fifty percent. On the other hand, the two Maxpool layers

[10]Opacus Github repository: https://github.com/pytorch/opacus

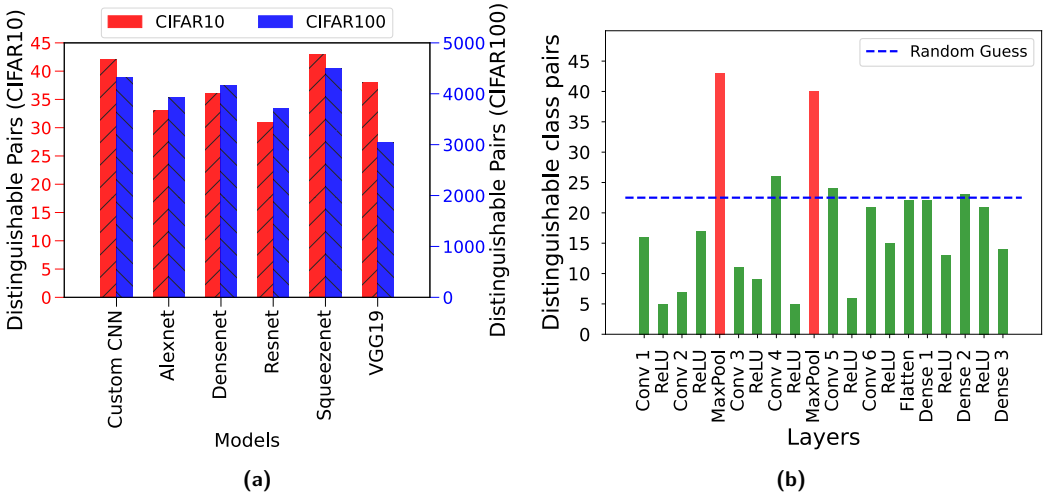

Figure 9: (a) Number of class pairs distinguishable by different CNN models trained with differential privacy on CIFAR10 (out of 45) and CIFAR100 (out of 4950) dataset (Intel Xeon processor)(b) Number of class pairs (out of 45) distinguishable in each layer of Custom CNN with Max Pooling trained with differential privacy(CIFAR10)

distinguish 43(95.5%) and 40(88.8%) class-pairs respectively. Hence, this again confirms the fact that Maxpool function is causing the timing difference, for differential-private CNNs as well. In the following discussion, we go one step further, launching the MLP class-label attack on the timing dataset of differential private custom CNN.

Attack on Differential Privacy Dataset:
We have verified the constant presence of PyTorch's vulnerability in differential privacy enabled CNN model from our timing analysis experiments. Our last step in this verification process is to check the effectiveness of our label classifier attack following the steps in Section 4. Our attack gives a high accuracy score of 99.2%, and we can now be sure that the noise added by Opacus does not have any impact on timing leakage. The test data we use has a total of 2000 data points, 200 for each class-label. Figure 10a shows the confusion matrix for the test data classification, which gives information about the *true label* of any class and also the *predicted label* of that class by the classifier. From the figure, we can see that for all classes we get more than 98% class-label classification accuracy. In the next section, we finally see how to leak information about the training set by bypassing differential privacy.

Table 3: Comparison of percentage of CIFAR10 and CIFAR100 class pairs distinguishable by different CNNs with and without differential privacy

Model	Overall Inference Accuracy			
	CIFAR10		CIFAR100	
	No DP	DP	No DP	DP
Custom CNN	93.33%	93.33%	85.29	87.35%
Alexnet	64.44%	73.33%	65.09	79.25%
Squeezenet	64.44%	80.0%	73.89%	84.26%
Densenet	73.33%	68.88%	71.03	75.15%
Resnet50	55.55%	95.55%	86.46%	90.70%
VGG19	75.55%	84.44%	55.29%	61.45%

5.2 Adversary Capabilities and Objective with DP-enabled MLaaS

The threat model is typically a real-life scenario of multiple clients accessing a cloud server that provides MLaaS where one amongst the many clients is considered to be adversarial, but this time the DL model on the server is trained with differential privacy which protects

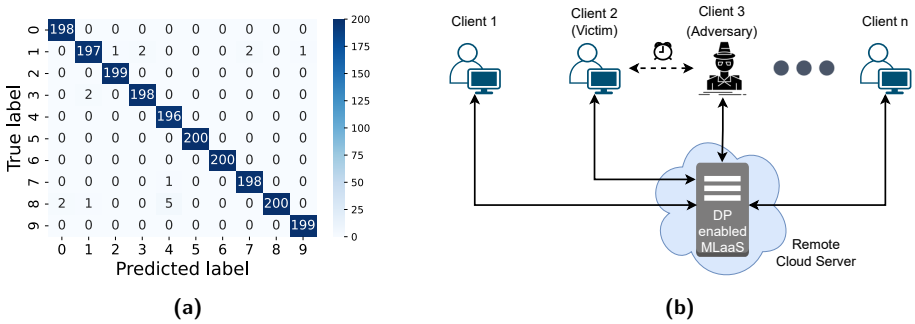

Figure 10: (a) Confusion Matrix for timing test data on the MLP model with timing values from custom CNN trained with DP (b) Threat Model for Privacy Violation in DP Models

training data information leakage (Refer to Figure 10b). Additionally, the private training data for the model is fed by the victim client accessing MLaaS, whereas the adversary tries to gain information about the training data.

The adversary connects with the victim client on the MLaaS server and has hard-label black-box access to the DL model with user-level privilege. The objective is to demonstrate the violation of Differential Privacy (DP) by illustrating the overlap of the dataset Q (the elements of which are mutually exclusive to the training set T) with a previously trained model, which should not be observed in an ideal DP scenario. The adversary infers the victim client's timing by the procedure as implied in Section 4.2. First, the adversary obtains the timing required for training with its own set, say T, and build the target model, say M_1. Subsequently, execution timing data of Q is observed by feeding it into the model M_1. At this point, the adversary adaptively updates M_1 by providing the inputs consisting of both T and Q and builds the model M_2 and similarly gets the timing data for inferring Q by model M_2. It will be interesting to observe that whether the timing data leads to the inference that Q has been used in the original training data T. It may be emphasized that by the definition of DP, any statistics (in our case timing) gathered by simulating with differential data (in our case T, and T augmented with Q), should not be distinguishable. Violation of this indicates a breach of the DP guarantees.

5.3 Countering Differential Privacy by Distinguishing Inputs

From previous discussions, it is established that the PyTorch vulnerability persists even after training DL models with differential privacy. In this section, we exploit this vulnerability for the purpose of bypassing differential privacy to mount an attack to breach training data's privacy. As illustrated in Figure 11, we divide the complete process into three parts:

1. *Train DP CNN Model with different training sets:* The adversary takes the set T and sends it to the victim client for training the MLaaS model using this dataset, to create *Model 1*. Next, the adversary takes the set Q and feeds it to *Model 1* for classification and the inference time to create the training and test datasets for the MLP classifier. Let us call the test set as *S1*. The adversary now sends the data Q to the victim for it to adaptively update the MLaaS model by training it with additional data, and create *Model 2*. Once again, the adversary collects the inference time of set Q using *Model 2* to create a new test set for the MLP classifier *S2*.
2. *Train MLP Classifier with Timing values from Model 1:* Now, we have one training set created with inference time values of Q from *Model 1* and two test sets created using timing values of Q from *Model 1* and *Model 2*. The adversary trains the MLP classifier

called *Label Classifier*, using the training set.

3. *Compare Timing results for Q with both models using MLP classifier:* Next, the adversary feeds both *S1* and *S2* to the *Label Classifier* separately and compares their classification accuracy. In our experiment, we take 1000 images (100 images of 10 class labels) in set Q and achieve an accuracy of 99.25% and 82.32% for *S1* and *S2*. We observe that the *S2* shows a reduced accuracy, indicating towards the fact that the inference times for set Q with *Model 1* and *Model 2* differ from each other. Next, we do an analysis for the case when the test data partially overlaps with the training data.

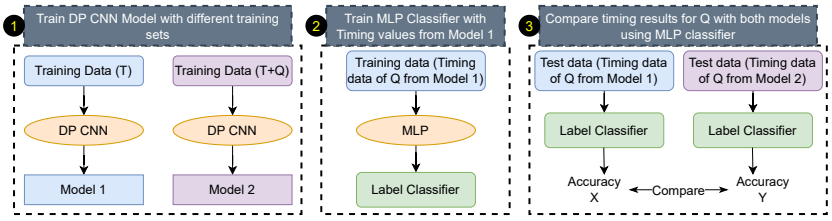

Figure 11: Bypassing Differential Privacy

Analysis with partial overlap of test and training data: In the previous experiment, we trained *Model 2* by adding the whole Q image set and observed a drop in accuracy of the MLP classifier which was trained with timing data from *Model 1*. Based on these results, we try to verify if this also happens when only a subset of images from set Q are added to the training dataset of *Model 2*. For our experiment, we start if 10% overlap of set Q with the training set and go on till 90% by increasing 10% overlap each time. The results are shown in Figure 12. We use the observation in Figure 12 as a yes/no test. If there is even a slight overlap between the test and the training dataset the accuracy drops. We do not see any increasing or decreasing trend in the plot, but the accuracy has dropped for by a minimum of 10% for all overlap ratios. We infer that even a slight amount of overlap of the test set with training set will reduce the accuracy of the classifier, which can be exploited by the adversary to launch an attack.

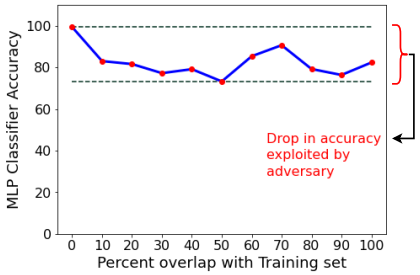

Figure 12: Accuracy of MLP Class-label classifier (trained using dataset with no overlap) over increasing ratios of training and test data overlap.

In general, we can confirm from all the results that the PyTorch vulnerability persists even after applying differential privacy to the models and can also be exploited to bypass differential privacy to leak information about the training dataset. In spite of differential privacy, Membership inference attack is also feasible with this vulnerability, hence we need a countermeasure to mitigate it. The best approach will be to mitigate it at the root level, by rectifying the Maxpool function's implementation, which we discuss in the next section.

6 Proposed Countermeasure against Class-Leakage

In the previous section, we explored in depth the timing vulnerabilities in PyTorch and developed attack methodologies to exploit them in a realistic framework. In this section, the vulnerability caused by the Maxpool function is thwarted by proposing an update to the existing implementation as a countermeasure. The idea is to patch PyTorch's `maxpool2d()` CPU function. PyTorch has multiple implementations of Maxpool functions (including `maxpool2d()`) for multiple types of input, devices, and applications, and this

countermeasure could easily be implemented at all places. This section begins by explaining the implementation of the proposed countermeasure in PyTorch, and later results of timing analysis and MLP attack, run on mitigated PyTorch library are discussed.

6.1 Countermeasure Implementation

It was shown in section 3.3 that Maxpool's implementation vulnerability is caused by the difference in the number of executions of assignment statements inside the 'if' condition. To mitigate this, we replicated the 'if' statement's functionality by adding a temporary swap location replacing the code snippet in Listing 1 with Listing 2. In the fixed code, an additional temporary array *tmp_arr* is introduced which is declared above the outermost *for* loop inside the *cpu_max_pool()* function. In Figure 13 we show that after implementing the countermeasure, assignment statement will be executed for every element of a window, hence the overall number of assignment operations inside the maxpool layer will be constant for all inputs. The yellow window represents the current window on which the Maxpool operation is being performed, and the bold green numbers indicate the positions at which the assignment operations are executed. For this example, we see that assignment operation is run for all windows hence constant time for all inputs. With the updated implementation, the input data dependency is removed completely, and therefore we claim to get uniform inference times for all classes, which are illustrated in the next section.

```
tmp_arr[0] = val;
tmp_arr[1] =  maxval;
maxval = tmp_arr[(val < maxval)*1];
tmp_arr[0] = index;
tmp_arr[1] = maxindex;
maxindex = tmp_arr[(val < maxval)*1];
```

Listing 2: Proposed constant time updated code to PyTorch Maxpool code structure

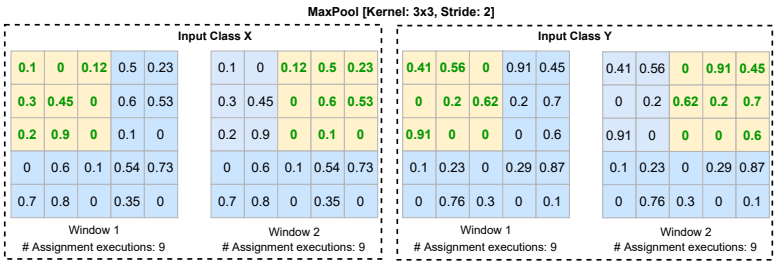

Figure 13: An example of constant number of assignment operations for two different inputs during Maxpool operation after implementing countermeasure

6.2 Mitigation of the PyTorch Vulnerability

In this section we explore the efficiency of the proposed countermeasure by repeating all the timing analysis experiments done in Section 3.2, and the experiments were repeated using the mitigation patched PyTorch library. We perform our experiments in a noiseless setting, when only one process is running the inferences, and can calculate its inference time from inside the process.

Analysing Overall Inference Time: Figure 14a shows timing analysis similar to Section 3.2.1 for all six models (Custom CNN, Alexnet, Resnet50, Densenet, Squeezenet and VGG19) implemented on countermeasure enabled PyTorch library.

Our decision criteria to distinguish class pairs is, when the number of distinguishable pairs is less than 50%, it means that the adversary has a random chance of guessing the

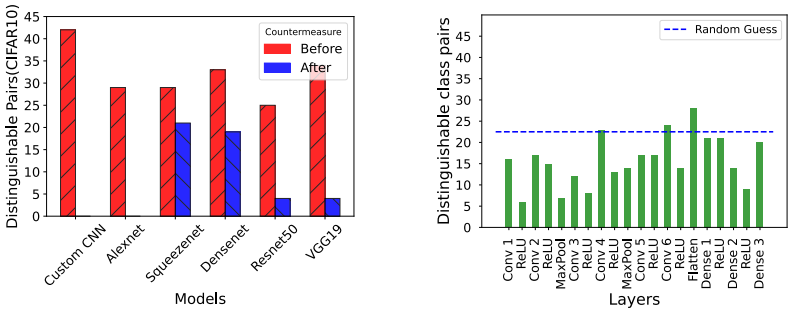

Figure 14: (a) Number of class pairs (out of 45) distinguishable by all CNN models before and after implementing the counter measure (CIFAR10) (b)Number of class pairs (out of 45) distinguishable in each layer of Custom CNN with Max Pooling after implementing the countermeasure (CIFAR10)

correct class labels. With the countermeasure, we observe that for all models with both CIFAR-10 and CIFAR-100 the number of distinguishable pairs is less than 50%. From this observation, we apparently make sure that the countermeasure implementation works as it claims to be. Next, we dig deep with layer-wise analysis, to see the timing behavior of the countermeasure on the exact source of leakage.

Analysing Layer-wise Inference Time: Having a working countermeasure we verify its effectiveness using the layer-wise analysis by following the experiment steps from Section 3.2.2. In Figure 14b we observe that the number of distinguishable pairs in the max pool layers has decreased to 7 and 14 from 43 and 40 in Figure 4(a). In the upcoming section, we verify the countermeasure's compatibility with differential-private networks.

Impact on model performance: The countermeasure implementation increases branching statement executions from before. Hence, there is a slight increase in the execution time (approximately 1-2 ms for a single inference) but no degradation in the model accuracy.

Cautionary Note: Our countermeasure only reduces the attack surface as other operations are still vulnerable albeit with more effort. Even constant time codes can be exploited in future with newer techniques like [WPH+22].

6.3 Mitigation of Class-Leakage in Differential Privacy

In Section 5.3, we demonstrated how differential privacy can be bypassed by exploiting Python's vulnerability, hence it becomes important to verify our countermeasure against the PyTorch models which are trained with differential privacy using the Opacus library.

Analysing Overall Inference Time: Once again we start by analysing the overall inference timing for all six differential-private models (Custom CNN, Alexnet, Resnet50, Densenet, Squeezenet, and VGG19). In Figure 15a, the number of distinguishable pairs once again reduces to less than 50% of the total pairs for all the models, hence confirming the effectiveness of our countermeasure for differential-private networks as well. Next, we explore the attack using MLP with countermeasure enabled in PyTorch.

6.4 Attack with Countermeasure

In Section 4.4 and Section 5.1 we saw that MLP class-label classifier gave high accuracy for custom CNN model trained with and without differential privacy. Now, in this section, we again launch the attack with implemented countermeasure.

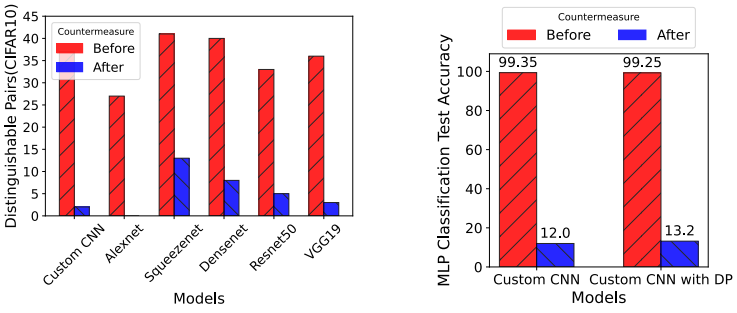

Figure 15: (a)Number of class pairs (out of 45) distinguishable by all CNN models trained with differential privacy before and after implementing the countermeasure (CIFAR10) (b)Accuracy of MLP model using inference time dataset of Custom CNN model

Attack on Custom CNN Dataset with Countermeasure: To confirm the effectiveness of our countermeasure, a new dataset for the MLP is created with the mitigated custom CNN model using the steps we discussed in Section 4. The MLP is trained with the dataset and then the classes of test data is inferred. Results show a tremendous drop in the accuracy from 99.35% to 12%, hence proving the viability of our countermeasure.
Attack on Differential Privacy Dataset with Countermeasure: Next follows the efficacy of our countermeasure on the dataset created using differential private custom CNN. We see a drop in accuracy from 99.25% to 13.2% which indicate towards random classification, meaning the model is no more able to classify among the different class label inputs. This shows that our countermeasure works perfectly in all scenarios. Figure 15b shows a comparison for attacks on both normal as well as differential private datasets.

7 Batch Inference Timing Analysis with Different Images

We have demonstrated in Section 4 how a label classifier can be developed using the observed timing leakage to mount an end-to-end attack even in a noisy scenario where multiple processes execute in the background. The attack involves the victim process running inference for the same image of a particular class 1000 times. In this section, we discuss a scenario where instead of running inferences for the same image multiple times, the victim can infer different images of a particular class only once in *batch inferences*. All the setup and other assumptions remain the same, as discussed before in Section 4.

7.1 Individual Timing Analysis on Batch Inferences

For this analysis, we select K images corresponding to each class from the CIFAR-10 dataset. We run individual inferences for all K images of each class and monitor their individual inference time. As a result, we obtain 10 timing distributions corresponding to each class, with each distribution consisting of K timing values. This analysis aims to inspect whether these 10 timing distributions are distinguishable from each other or not. We consider $K = 2000$ for our experiment. Figure 16a shows the frequency plots for the 10 distributions formed using 2000 timing values from each class. The vertical dotted lines indicate the mean of the corresponding distribution. We observe that the distributions are distinct with different mean values, demonstrating the existence of data-dependent timing leakage for different classes, even if we consider different images, due to the non-constant time implementation of the MaxPool operation.

Timing Analysis in Noisy Setup: The previous analysis shows the existence of timing leakage in a noiseless setup, i.e., when no other process runs in parallel while the inferences are performed. Next, we perform the same analysis in a noisy setup with a total of n processes running simultaneously, including the victim process. We consider two scenarios for $n = 4$ and $n = 8$. In each scenario, we consider the batch size of 2000 images for each class. We show their frequency plots in Figure 16b and Figure 16c, respectively. We observe that even in the noisy setup, the distributions of the 10 classes are still distinct, and the order of class-timings is the same as what we observed in Figure 16a. From Figures 16a, 16b, 16c we infer that all images of a given class have got related timings, hence we see ten different clusters for the ten classes of CIFAR-10. The leakage from multiple images inferred in batches can be exploited to launch an end-to-end attack to obtain the class label of a particular batch of inputs. Next, we analyze the consistency of multi-image batch inference timings by ranking each class based on their cumulative timing from all images in the batch.

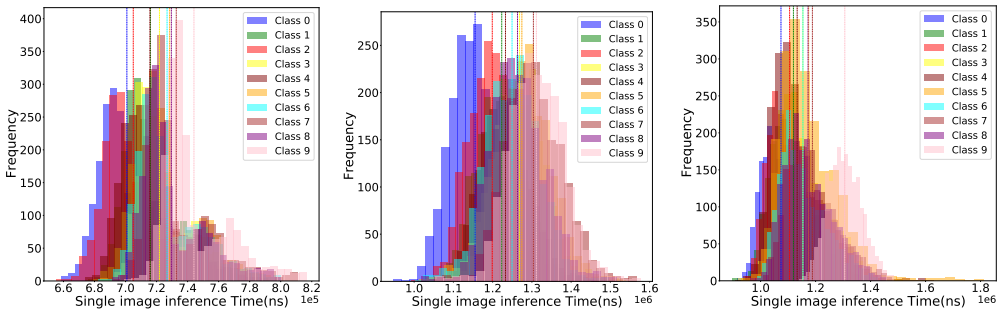

Figure 16: Frequency plots for 2000 CIFAR-10 image inference timings for 10 classes with (a) no parallel process, (b) 4 parallel processes and (c) 8 parallel processes

7.2 Cumulative Timing Analysis on Batch Inferences

The analysis in the previous subsection requires individual inference timings of $K = 2000$ images which are challenging in practice, especially in batch inferences. In addition, the noise from other background processes may also affect the individual timing values. Hence, for this analysis, we consider the cumulative sum of inference times for all images in a batch instead of individual timing values. We obtain a vector of dimension 10 corresponding to each class of images. Further, we sort the timing values in the vector in ascending order to get the ranking of each class for creating a timing template for the classes. Figure 17 shows the consistency of these class ranks with two separate experiments. First, we select a batch with $K = 2000$ images for each class and repeat the batch inference for the chosen batch 10 times. We compute the sum of individual inference times of each image for a particular batch in each run, resulting in a total of 10 vectors for 10 different runs. The ranks of batches belonging to different classes for each run based on the cumulative timing value are shown at the top of Figure 17. The purple color indicates the ranks which are not consistent.

We observe a consistent ranking 80% (80/100) of the time. In the next experiment, we select 4 different batches of 2000 images at each run instead of repeating the batches multiple times and perform a similar ranking analysis as the first experiment. We obtain four vectors of dimension 10 corresponding to four different runs. The ranks of batches belonging to different classes for each run are shown at the bottom of Figure 17. We

observe a consistent ranking 80% (32/40) of the time in this experiment as well, similar to Figure 17. The few inconsistent rankings are for classes with closely overlapped timing clusters, as seen in Figure 16a.

8 Vulnerability Disclosure

We reported the observed data-dependent timing side-channel leakage due to improper non-constant time implementation of Max Pooling operation in PyTorch to Meta (Facebook) AI research (developer of the PyTorch library) on 16 January 2022. We received an acknowledgment for reporting a valid issue on 14 February 2022. Quoting their exact words: *"We have discussed the issue at length and concluded that, whilst you reported a valid issue which the team may make changes based on, unfortunately your report falls below the bar for a monetary reward."* We further reported the practical attack on user privacy exploiting the data-dependent timing side-channel leakage, the vulnerability of differential private deep learning models against membership inference attacks exploiting the same timing-channel, and the countermeasure to alleviate the data-dependent timing side-channel.

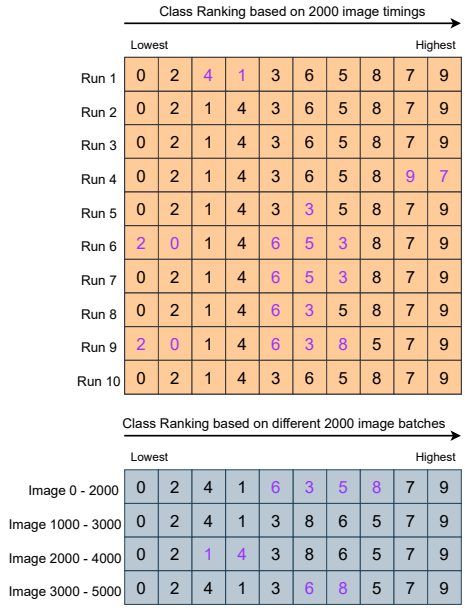

Figure 17: (top) Rank analysis of classes over 10 runs after summing the timing of 2000 images of each class label. (bottom) Rank analysis of classes with 4 different subsets of 2000 images from each class.

9 Conclusion

In this paper we bring forward for the first time, a potential privacy threat found in the PyTorch library caused by timing side-channel leakage, which an adversary can exploit to get the input class of the data being fed to a neural network. The source of leakage was diagnosed to be the Maxpool layer of the network. The vulnerability caused by this leakage can be exploited to classify class labels of the inputs by training a MLP classifier with statistically processed inference time dataset. This was further utilized using the MLP classifier to bypass differential privacy to identify whether a set of inputs to the model belongs to CNN model's training dataset. Finally, we propose an inexpensive yet effective implementation as a countermeasure to thwart such timing vulnerability. Our current attack only works when the victim infers inputs from the same class hence for future work, we would like to extend our attack for the scenario where the victim runs inferences of different classes.

Acknowledgements

We thank Prof. Maria Méndez Real and the other anonymous reviewers of TCHES for their valuable feedback and comments. This research was supported in part by the Prime Minister's Research Fellowship, provided by Ministry of Education, Government of India. We would also like to thank Defence Research & Development Organisation (DRDO),

Government of India for supporting publication under the project "Secure Resource-constrained Communication Framework for Tactical Networks using PUFs (SeRFPUF)".

References

[A+16a] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[A+16b] Rami Al-Rfou et al. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[ACG+16] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 308–318. ACM, 2016.

[AM19] Manaar Alam and Debdeep Mukhopadhyay. How secure are deep learning algorithms from side-channel based reverse engineering? In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 226. ACM, 2019.

[AMAZ17] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[Bal20] Sameer Balaganur. How pytorch is increasingly being adopted by organisations. https://analyticsindiamag.com/how-pytorch-is-increasingly-being-adopted-by-organizations/, February 2020.

[BBJP19] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: reverse engineering of neural network architectures through electromagnetic side channel. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 515–532. USENIX Association, 2019.

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[Car20] Scott Carey. Why enterprises are turning from tensorflow to pytorch. https://www.infoworld.com/article/3597904/why-enterprises-are-turning-from-tensorflow-to-pytorch.html, December 2020.

[CTCP21] Christopher A. Choquette-Choo, Florian Tramèr, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 1964–1974. PMLR, 2021.

[DB17] Murat Dikmen and Catherine M. Burns. Trust in autonomous vehicles: The case of tesla autopilot and summon. In *2017 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2017, Banff, AB, Canada, October 5-8, 2017*, pages 1093–1098. IEEE, 2017.

[DSRB18] Vasisht Duddu, Debasis Samanta, D. Vijay Rao, and Valentina Emilia Balas. Stealing neural networks via timing side channels. *CoRR*, abs/1812.11720, 2018.

[GDL+16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016.

[GK20] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. *CoRR*, abs/2009.07485, 2020.

[GTCM20] Sorin Mihai Grigorescu, Bogdan Trasnea, Tiberiu T. Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *J. Field Robotics*, 37(3):362–386, 2020.

[HDK+18] 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.

[HLvdMW17] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[HZS18] 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 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 4:1–4:6. ACM, 2018.

[IMA+16] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[KH+09] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[KHL11] Damir Kalpic, Nikica Hlupic, and Miodrag Lovric. Student's t-tests., 2011.

[KPPS21] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2651–2668. USENIX Association, 2021.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

[KVH+21] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *CoRR*, abs/2109.00984, 2021.

[LJ19] Qian Lou and Lei Jiang. SHE: A fast and accurate deep neural network for encrypted data. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10035–10043, 2019.

[Mej19] Niccolo Mejia. Artificial intelligence at mckesson - ai initiatives and investments, July 2019.

[MTH+21] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. Remote power side-channel attacks on bnn accelerators in fpgas. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1639–1644. IEEE, 2021.

[NNQA18] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael B. Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2139–2153. ACM, 2018.

[NSF21] Tsunato Nakai, Daisuke Suzuki, and Takeshi Fujino. Timing black-box attacks: Crafting adversarial examples through timing leaks against dnns on embedded devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):149–175, 2021.

[NSH19] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 IEEE symposium on security and privacy (SP)*, pages 739–753. IEEE, 2019.

[NSH20] Milad Nasr, Reza Shokri, and Amir Houmansadr. Improving deep learning with differential privacy using gradient encoding and denoising. *CoRR*, abs/2007.11524, 2020.

[P+19] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[Pat19] Luigi Patruno. Batch inference vs online inference. https://mlinproduction.com/batch-inference-vs-online-inference/, March 2019.

[PMG+17] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[PTS+21] Nicolas Papernot, Abhradeep Thakurta, Shuang Song, Steve Chien, and Úlfar Erlingsson. Tempered sigmoid activations for deep learning with differential privacy. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 9312–9321. AAAI Press, 2021.

[PVG+12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *CoRR*, abs/1201.0490, 2012.

[Ray17] Alison DeNisco Rayome. Machine learning as a service to hit nearly $20b by 2025, driven by healthcare and life sciences, April 2017.

[RCYF21] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. *CoRR*, abs/2111.04625, 2021.

[RGC15] Mauro Ribeiro, Katarina Grolinger, and Miriam A. M. Capretz. Mlaas: Machine learning as a service. In *14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, December 9-11, 2015*, pages 896–902. IEEE, 2015.

[SSSS17] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 3–18. IEEE Computer Society, 2017.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[TIM] Python time library. https://docs.python.org/3/library/time.html.

[TLG+21] Stacey Truex, Ling Liu, Mehmet Emre Gursoy, Lei Yu, and Wenqi Wei. Demystifying membership inference attacks in machine learning as a service. *IEEE Trans. Serv. Comput.*, 14(6):2073–2089, 2021.

[VAS+19] R. Vinayakumar, Mamoun Alazab, K. P. Soman, Prabaharan Poornachandran, and Sitalakshmi Venkatraman. Robust intelligent malware detection using deep learning. *IEEE Access*, 7:46717–46738, 2019.

[Wan19] Angela Wang. Parallelizing across multiple cpu/gpus to speed up deep learning inference at the edge. https://aws.amazon.com/blogs/machine-learning/parallelizing-across-multiple-cpu-gpus-to-speed-up-deep-learning-inference-at-the-edge/, August 2019.

[WCJ+21] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Shivam Bhasin, and Arindam Basu. Time to leak: Cross-device timing attack on edge deep learning accelerator. In *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4. IEEE, 2021.

[WHP+] Han Wang, Syed Mahbub Hafiz, Kartik Patwari, Chen-Nee Chuah, Zubair Shafiq, and Houman Homayoun. Stealthy inference attack on dnn via cache-based side-channel attacks.

[WLL+18] 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 2018, San Juan, PR, USA, December 03-07, 2018*, pages 393–406. ACM, 2018.

[WPH+22] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 679–697, 2022.

[WZZ+20] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky DNN: stealing deep-learning model secret with GPU context-switching side-channel. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 125–137. IEEE, 2020.

[XCC+20] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open DNN box by power side-channel attack. *IEEE Trans. Circuits Syst.*, 67-II(11):2717–2721, 2020.

[YFT20] 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 2020, August 12-14, 2020*, pages 2003–2020. USENIX Association, 2020.

[YKSF19] Kota Yoshida, Takaya Kubota, Mitsuru Shiozaki, and Takeshi Fujino. Model-extraction attack against FPGA-DNN accelerator utilizing correlation electromagnetic analysis. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*, page 318. IEEE, 2019.

[YMY+20] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. Deepem: Deep neural networks model recovery through EM side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 209–218. IEEE, 2020.

[YSS+21] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Gosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. Opacus: User-friendly differential privacy library in pytorch. *CoRR*, abs/2109.12298, 2021.