NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2023-05

April, 2023

Benchmarking the Overhead of Running Neural Networks in OP-TEE

Peizhi Liu Huaxuan Chen Zhenguo Mo Peter Dinda

Abstract

OP-TEE is a secure Trusted Execution Environment (TEE) designed to run alongside the normal OS on ARM platforms. Trusted Applications in OP-TEE, while offering many advantages to securely process information, are not well suited to support deep learning due to very limited memory constraints. To date, there have been many proposed methods to limit the size of deep-learning models. In this paper, we explore the overhead of running a fully connected neural network in the TEE. We also analyze the performance tradeoff of a layer-based partitioning method used to reduce the memory footprint of deep learning models. As a means to benchmark the performance of Trusted Applications, we implement a low-overhead benchmarking framework across the Normal world and Secure world via the system call interface. Our results reveal that while the TEE itself presents limited overhead, layer-based partition brings about a significant slowdown to the model's computation time while resulting in only marginal benefits to typical memory consumption.

Keywords

OP-TEE, neural networks, layer-based partitioning, trusted application

Benchmarking the Overhead of Running Neural Networks in OP-TEE

Peizhi Liu
Northwestern University

Huaxuan Chen
Northwestern University

Zhenguo Mo
Northwestern University

Peter Dinda
Northwestern University

ABSTRACT

OP-TEE is a secure Trusted Execution Environment (TEE) designed to run alongside the normal OS on ARM platforms. Trusted Applications in OP-TEE, while offering many advantages to securely process information, are not well suited to support deep learning due to very limited memory constraints. To date, there have been many proposed methods to limit the size of deep-learning models. In this paper, we explore the overhead of running a fully connected neural network in the TEE. We also analyze the performance tradeoff of a layer-based partitioning method used to reduce the memory footprint of deep learning models. As a means to benchmark the performance of Trusted Applications, we implement a low-overhead benchmarking framework across the Normal world and Secure world via the system call interface. Our results reveal that while the TEE itself presents limited overhead, layer-based partition brings about a significant slowdown to the model's computation time while resulting in only marginal benefits to typical memory consumption.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; • **Computing methodologies → Neural networks**.

KEYWORDS

OP-TEE, neural networks, layer-based partitioning, trusted application

1 INTRODUCTION

The deployment of Deep Neural Networks (DNNs) on edge devices such as smartphones has been made possible as the memory and processing resources improve and advance. However, as DNNs become more popular on edge devices, privacy risks arise from the fact that DNN models are not encrypted or secured. For a pre-trained model stored on an edge device, the confidential and private information stored in a model can be exploited by Membership Inference Attacks (MIAs). These attacks can have severe privacy consequences that have already motivated researchers to develop and deploy unique and innovative systems to counterattack.

Open Portable Trusted Execution Environments (OP-TEE) is an open-source Trusted Execution Environments (TEE) implementing the Arm TrustZone technology. OP-TEE can run as a companion to a non-secure Linux operating system to allow applications to run inside the TEE. These applications are referred to as Trusted Applications (TAs) and they are isolated and protected by the TEE from the non-secure main operating system. Based on this secure execution environment, it is possible to hide network layers in the TEE. In fact, it is necessary in many situations to store the model fully inside the TEE to isolate all sensitive data from the non-secure environments, such as the main operating system, hence the attacks. This optimal approach theoretically addresses privacy issues.

While TAs offer security from the real execution environment (REE), the trusted applications in OP-TEE are, however, severely limited both in terms of memory as well as performance, thus making it not particularly suitable for Deep Neural Networks. In fact, it is estimated that the maximum default TA size on the Raspberry Pi 3b is 30 MB [1]. That said, the performance of deep learning models stored and running in a secure but limited environment is a topic worth investigating. Furthermore, various methods of model size reduction in the TA are also an area of active research [7]. In particular, in this paper, we will focus on layer-based partitioning as a means to reduce the model size, as well as the performance impacts of such an approach.

In Section 2, we will first discuss in more detail the existing work to run DNNs in a secure and trusted environment and the context surrounding them. We will then delve into the specifics of the fully connected neural network with layer-based partitioning we implemented for the project in Section 3. Section 4, will explore the motivation and implementation of a framework to benchmark neural-network performance in the TA. In Section 5, we set up and perform experiments to answer the aforementioned questions on neural-network performance in TAs. We then analyze the results in Section 6 and summarize our findings in Section 7.

This project was supported by the United States National Science Foundation via grant CNS-2211508.

2 BACKGROUND

2.1 OP-TEE

Open Portable Trusted Execution Environment (OP-TEE) is a Trusted Execution Environment (TEE) that implements an API for running trusted applications. It is a companion operating system (OS) running alongside a non-secure OS. The non-secure OS is also called Rich Execution Environment (REE) where it communicates with TEE through TEE Client API. The TEE is for running and storing Trusted Applications (TA). TAs are isolated and protected by the TEE from the REE. For a platform without hardware security measures, OP-TEE is able to provide the security it lacks. However, OP-TEE has very limited memory, only around 30 MB on Raspberry Pi 3B is available to trusted applications by default [1], limiting the utility of TA.

2.2 Existing Works

Running machine learning models in a trusted environment is nothing new. In fact, layer-based partition itself in OP-TEE was first proposed by VanNostrand et. al. [7] along with other partitioning methods such as sub-layer partitioning where each layer in a DNN model is further divided into sub-layers to perform the computation, as well as branched partitioning where layers in a DNN model can be vertically partitioned into separate, mutually independent sub-networks.

Aside from methods to reduce network size in the TEE, there have been various works that implement neural networks that run partially in trusted applications [5][4]. In particular, DarkneTZ is a deep learning framework that builds on top of the Darknet [6] deep learning framework to train and perform inference using deep learning models in a trusted application environment. In particular, the trusted application has the ability to run several layers of a DNN model in the TrustZone, and it can be used to train several types of DNNs, such as VGG-7, AlexNet, Resnet-50, with layers in both the REE and TEE. Furthermore, like Darknet, DarkneTZ can also load pre-trained models for inference.

While DarkneTZ provides a convenient interface to train and execute deep learning models that are only partially loaded inside a trusted execution environment, it, however, lacks the function to dynamically load layers in and out of storage in the TA during training and inference. As such, we decided against using DarkneTZ for the implementation of the neural network and implemented our own neural network detailed in the next few sections.

3 LAYER-BASED PARTITIONING

To reduce memory usage of neural networks in trusted applications, layer-based partitioning breaks up the network into individual layers where only a single layer may be loaded into the TA at any given time while the other unused layers will be stored encrypted in the real execution environment until it is needed, where the encrypted layer will then get passed into the TA, decrypted, and deserialized. The process of storing and loading a neural network, especially fully connected layers in a neural network, naturally lends itself to this type of partitioning as inputs and outputs of a single layer come only from the outputs of the previous layer and act as the inputs to the next layer respectively [8].

In this paper and in our implementation, we will further generalize the idea of layer-based partitioning to multiple layers. That is, multiple layers in the network can be grouped together to act as a single partition stored and loaded into the TA as needed.

3.1 Fully Connected Neural Networks

Our neural network implementation has a special focus on fully connected neural networks (FCNNs) as they often appear in other network models and intuitively allow for layer-based partitioning. FCNNs have a topology where each neuron in a layer is connected to every neuron in the following layer. The input of the FCNN takes in features and performs a feature transformation to obtain outputs. By tuning weights and biases at each layer in the network, or in the feature transformation, we can change how the input space gets mapped to the output space.

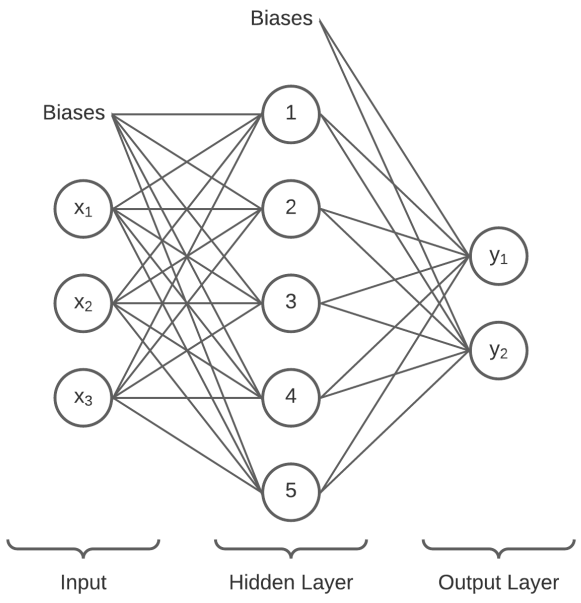

Figure 1: Example of a fully connected neural network with an input (3 neurons), an output layer (2 neurons), and a single hidden layer (5 neurons).

As an example, consider figure 1 where we have a simple network composed of an input and output layer, as well as a

single hidden layer. The output of each neuron in the hidden and output layer are functions of all values of neurons in the previous layer, transformed by weights located along the edges connecting the neurons and a bias. For instance, the output of neuron 3 in the hidden layer of the shown FCNN can be expressed as

$$f_3^{(1)}(\mathbf{x}) = a(w_{3,1}^{(1)}x_1 + w_{3,2}^{(1)}x_2 + w_{3,3}^{(1)}x_3 + b_3^{(1)}) \quad (1)$$

With a as the activation function and where the superscripts on the transformation, weights, and biases represent the layer number. The first subscript on the same terms represents the neuron in the current layer, and the second subscript, if one exists, represents the neuron from the previous layer. In general, given an $L - layer$ FCNN, we can express the output of the nth neuron in the L^{th} layer using the same notation as above to be

$$f_3^{(L)}(\mathbf{x}) = a(b_n^{(L)} + \sum_{i=1}^{U_{L-1}} w_{n,i}^{(L)} f_i^{(L-1)}(\mathbf{x})) \quad (2)$$

where U_{L-1} is the number of neurons in the previous layer and consequently, U_L is the number of neurons in the current layer L. The expression above can also be compactly written in matrix notation

$$a(b_n^{(L)} + \sum_{i=1}^{U_{L-1}} w_{n,i}^{(L)} f_i^{(L-1)}(\mathbf{x}) = a([\mathbf{b}^{(L)} + \mathbf{W}_L^T \mathbf{f}^{(L-1)}(\mathbf{x})]_n) \quad (3)$$

in which $\mathbf{W}_L$ is a $U_{L-1} \times U_L$ matrix,

$$\mathbf{W}_L = \begin{bmatrix} w_{1,1}^{(L)} & w_{2,1}^{(L)} & \dots & w_{U_L,1}^{(L)} \\ w_{1,2}^{(L)} & w_{2,2}^{(L)} & \dots & w_{U_L,2}^{(L)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,U_{L-1}}^{(L)} & w_{2,U_{L-1}}^{(L)} & \dots & w_{U_L,U_{L-1}}^{(L)} \end{bmatrix} \quad (4)$$

$\mathbf{b}^{(L)}$ is a $U_L \times 1$ vector, and $\mathbf{f}^{(L-1)}$ is a $U_{L-1} \times 1$ vector.

$$\mathbf{b}^{(L)} = \begin{bmatrix} b_1^{(L)} \\ b_2^{(L)} \\ \vdots \\ b_{U_L}^{(L)} \end{bmatrix}, \mathbf{f}^{(L-1)} = \begin{bmatrix} f_1^{(L-1)} \\ f_2^{(L-1)} \\ \vdots \\ f_{U_{L-1}}^{(L-1)} \end{bmatrix} \quad (5)$$

3.2 Secure Storage

OP-TEE implements GlobalPlatform Technology's TEE Internal Core Trusted Storage API via secure storage. The purpose of such an API is to allow users to store general-purpose data objects as well as cryptographic objects securely and to allow users to modify stored data atomically [3][2]. OP-TEE actually has two implementations of secure storage. The first makes use of the Linux file system in the REE while the second uses the Replay Protected Memory Block (RPMB) partition of eMMC storage to store data [3]. In our implementation, we stored the serialized layer as data objects using the REE file system secure storage implementation.

Table 1 shows the trusted storage API endpoints that were used to store and retrieve layer data. We designed the storage of layers such that every layer in an FCNN will have its corresponding layer object in secure storage, identified by the layer number. For example, layer 0 will have a corresponding layer object with ID 0x00000000. The corresponding layer objects are first created during network initialization and future retrieval and modification operations will occur during forward propagation and backpropagation.

Future modifications such as storing multiple layers in one secure storage object can perhaps be made to more efficiently retrieve and store layers when multiple layers are part of a single partition. Now, instead of an object for every layer, there will only be an object for every partition of the network, reducing the number of trusted storage API calls.

3.3 Layer Serialization

Layers in the network must be stored and retrieved by writing to and reading from a shared buffer given to the trusted storage API from the TA. However, in our implementation, layers are represented as C structures of integers and matrices. This requires us to serialize and deserialize layers to and from byte arrays. These byte arrays will then act as buffers for storing and reading layer data objects to and from the secure storage.

To perform the marshaling and unmarshalling of layer structures, we first defined a serialization scheme for different fields, integers, and matrices, in a layer structure. As shown in figure 2, serialization for integers contains only the type metadata followed by the integer value, while, also in figure 2, serialization for matrices contains additional metadata/data about the dimensionality of the matrix. The dimensionality of the matrix must be known before reading the values of the matrix to prevent overflowing the buffer and to help with initializing a new matrix. Using these two primitive serialization schemes, all the fields of a layer can be marshaled into a single-byte array suitable for storage and retrieval.

3.4 Forward Propagation with Layer-Based Partitioning

Forward propagation of input data through the network occurs both during training as well as during inference. Using the same notation as described in section 3.1, forward propagation of an $L - layer$ fully connected neural network can be written compactly as

$$model(\mathbf{x}, \Theta) = a(\mathbf{b}^{(L)} + \mathbf{W}_L^T a(\dots a(\mathbf{b}^{(1)} + \mathbf{W}_1^T \mathbf{x}))) \quad (6)$$

Table 1: Trusted Storage API endpoints used to implement layer-based partitioning [2]

Endpoint	Use
`TEE_CreatePersistentObject`	Called each time to store a serialized layer from a buffer in the TA. This endpoint is also used to delete previously stored layer information.
`TEE_OpenPersistentObject`	Obtain a persistent layer object handle.
`TEE_GetObjectInfo1`	Retrieve stored size of persistent layer object.
`TEE_ReadObjectData`	Read serialized layer object data to a buffer in the TA.
`TEE_CloseObject`	Close the layer object handle.

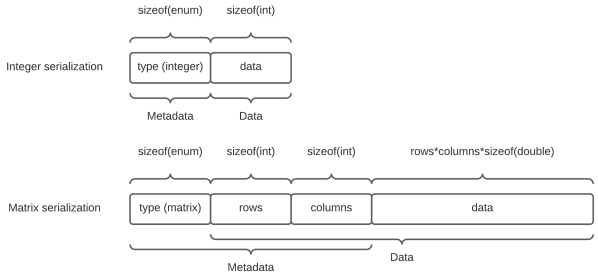

Figure 2: Serialization of integers and matrices in C.

Depicted from the expression and as noted above, outputs of one layer become inputs of the next layer. The result of this is that layer-based partitioning of the FCNN is relatively straightforward as we can load subsequent layers one after the other into TA memory. When a single partition involves n layers, forward propagation will attempt to iterate through all layers in batches of n-sized partitions. It is entirely possible for the last partition to have less than n layers. In that case, the last partition loaded into the TA will just have the remaining layers.

Aside from this, it is important to note that the outputs of one layer partition must be stored temporarily before getting fed as inputs to the subsequent layer partition after it is retrieved from storage. Consequently, the minimum memory amount required by the TA to run the model must also take into account the dimensionality of input data and mini-batch size.

3.5 Backpropagation with Layer-Based Partitioning

The main goal of backpropagation is to find the gradient of the cost function with respect to the weights of the network. With the gradient, we are able to train and optimize the model through gradient descent in an iterative fashion. In particular, the gradients of the cost with respect to the weights and biases of a single layer L for a fully connected neural network can be expressed as the following

$$\nabla_{\mathbf{W}_L} C = (\mathbf{f}^{(L-1)}(\mathbf{x}))^T \delta^{(L)} \tag{7}$$

$$\nabla_{\mathbf{b}^{(L)}} C = \delta^{(L)} \tag{8}$$

where C is the cost function and $\delta^{(L)}$ is

$$\delta^{(L)} = a'(\mathbf{f}^{(L)}(\mathbf{x})) \odot \begin{cases} \delta^{(L+1)} \mathbf{W}_{L+1}^T, & 0 < L < N \\ C'(\mathbf{f}^{(L)}(\mathbf{x}), \mathbf{y}), & L = N \end{cases} \tag{9}$$

in which $\mathbf{x}$ is the feature, $\mathbf{y}$ is the label, and N is the total number of layers in this network.

As seen from the above expression for computing the gradients, backpropagation requires us to iterate through the layers in reverse order. In terms of layer-based partitioning, this would mean we have to load layer partitions into the TA in reverse order. For example, suppose we have an FCNN with three layers in total but specified we only want two layers loaded into the TA at once. This would mean the last two layers will first be loaded into the TA as a single partition, followed by the first layer when computing the gradient of the cost w.r.t. the weights and biases in layer one.

Furthermore, as noted in the above expressions for computing gradients, we must also store the input and output of each layer in the layer structure during forward propagation for reference to use in backpropagation. In addition, we also store $\nabla_{\mathbf{W}_L} C$ and $\nabla_{\mathbf{b}^{(L)}} C$ in the layer struct for use during gradient descent to update the weights. Overall, the memory footprint of the network is not simply the weights and biases of the network, but also must take into consideration additional fields if one wants to train the networks inside the TEE.

4 BENCHMARKING

To collect performance data of DNNs in trusted applications as well as to measure the induced overhead of layer-based partitioning, we wrote a custom benchmarking framework

Table 2: Benchmark Framework API

Endpoint	Use
TEE_InitSctrace	Sets initial benchmark context.
TEE_AddSctrace	Adds new trace to existing traces.
TEE_GetSctrace	Writes all existing traces to a CSV file.
TEE_ResetSctrace	Clears the existing traces.

similar to [1] that leverages system calls from both the REE and the TEE to add, get, and reset performance traces.

4.1 Motivation

We needed a benchmarking framework that gives us very fine control over the portion of code in the TEE that we want to measure the performance. Furthermore, since we are also measuring the memory footprints of the network and memory is valuable in trusted applications, it is a good idea to store the trace history outside of the TEE and in the REE. As a result, trace data from within the TA must be forwarded via a series of system calls and RPCs into the normal world.

4.2 Design

The overall implementation of the benchmark has four functions that were appended to OP-TEE's global API as noted in table 2. `TEE_InitSctrace` initializes the values in the benchmarking context, `TEE_AddSctrace` is called from within a TA with an integer ID to get the time delta and amount of allocated memory in the TA. `TEE_GetSctrace` takes in an integer once again and stores all existing traces as a CSV file in the REE on the Linux file system. The prefix is set to the integer passed in. Note that `TEE_GetSctrace` does not reset the existing traces in the REE, but the overhead of this call is accounted for. Finally, `TEE_ResetSctrace` is used to clear all existing traces in the REE.

As mentioned before, trace values in the TA are sent to the REE and stored via a series of system and RPC calls. In particular, when a `TEE_<Verb>Sctrace` function (except `TEE_InitSctrace`) gets invoked, it will first make an OP-TEE system call. The OP-TEE system call will construct and make an RPC call to the OP-TEE driver in Linux. Upon receiving the RPC message, the driver, which is now in Linux kernel mode, will invoke a Linux system call to pass in the received data and add a new trace entry to the existing traces already in the REE.

Existing traces are stored in the REE as a global variable in the form of a doubly-linked list of `sctrace_t` structures:

```
typedef struct sctrace
{
    unsigned long id;
    unsigned long delta;
    unsigned long allocated;
    long ree_time;
    struct sctrace* next;
    struct sctrace* prev;
} sctrace_t;
```

The field `id` is the value of the parameter passed into the `TEE_AddSctrace` function, `delta` is the time in microseconds between the previous and current `TEE_AddSctrace` calls, `allocated` is the number of bytes currently allocated in the TA, and `ree_time` is the current time of the REE in nanoseconds.

4.3 Overhead

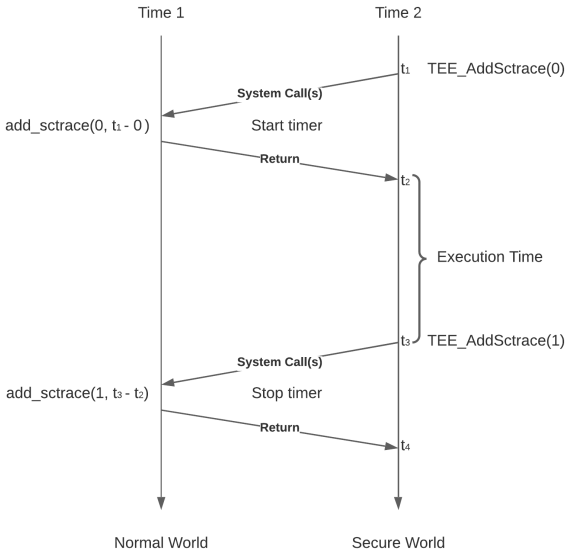

Figure 3: Using system calls to store traces in the normal world adds additional overhead from the benchmark. To account for such overhead, we calculate the execution time between `TEE_AddSctrace` calls as the delta between the start of the current call and the end of the previous call.

The use of system calls to pass traces from within the TEE to the REE is useful for isolating the memory used by the benchmark itself. However, system calls that we invoke to add traces will take time to complete both in OP-TEE and in Linux, resulting in additional time overhead. As alluded to before in section 4.2, we use time deltas in the TA to address this issue. Consider figure 3 which depicts the overall sketch of the problem and solution. To account for the issue of additional overhead from `TEE_AddSctrace`, we can calculate the time between such calls as the difference between the

start of the current call and the end of the previous call. This forces us to use TEE_AddSctrace in a particular way. Suppose we wanted to find the execution time of a block of code. We will first need to surround that block with two TEE_AddSctrace calls. Then to read the execution time, we must look at the time delta of the second TEE_AddSctrace call. While this seems complicated at first, we can also easily integrate over the time deltas in a list of trace entries to get the actual time. Using the integrated times, we can simply subtract the times at the first add trace call from the time at the second add trace call to get the execution time.

The actual implementation of the time delta calculation is done via a global time_context_t structure in the TEE:

```
typedef struct time_context
{
    uint64_t start;
    uint64_t end;
    uint64_t overhead;
} time_context_t;
```

The start field stores the start time of a TEE_AddSctrace call while the end field stores the time right before a call to TEE_AddSctrace returns. overhead is an added field to account for the overhead of calling TEE_GetSctrace in between TEE_AddSctrace calls. All fields in the time_context_t structure are in units of microseconds.

As briefly mentioned before, TEE_InitSctrace is initially called, it sets the value of time_context_t.end to be the current time. Afterward, whenever TEE_AddSctrace is invoked, it will first set time_context_t.start to be the current time. Next, it will calculate the time delta as

$$delta = start - end - overhead \tag{10}$$

While we have accounted for the major overheads of sending traces via system calls, there is yet another system call that we did not account for. That is, the syscall to obtain the current time in microseconds in OP-TEE. We, however, measured the additional overhead of this syscall to be quite small at around 5.5μs on average.

5 EXPERIMENTAL SETUP

5.1 Platform

We chose to run our experiments on the Raspberry Pi 3 Model B due to an abundant array of related literature that already uses this platform [7][1][4]. Furthermore, there has been a thorough performance characterization of OP-TEE on the Pi 3B [1]. As mentioned before, performance and available memory in the TA is quite limited. In terms of performance, trusted applications do not have multi-threading. Thus, our FCNN implementation runs on only one thread in the TA. Considering memory, the memory size set aside for trusted applications on the platform is quite small defaulting at 30 MB. However, we were able to expand the heap size of the TAs with the largest we tested being 250 MB. In this experiment, however, we configure the TA heap size to be 60 MB and the stack size to be 1 MB.

Table 3: OP-TEE memory-related settings changed.

Setting	Value
CFG_SHMEM_SIZE	0x02000000
CFG_TZDRAM_SIZE	0x04000000
CFG_TEE_RAM_VA_SIZE	0x00200000
PGT_CACHE_SIZE	70
TA_STACK_SIZE	(60 * 1024 * 1024)
TA_DATA_SIZE	(1 * 1024 * 1024)

Increasing the heap size of the TA can be done by configuring a higher trust-zone DRAM size via CFG_TZDRAM_SIZE, changing CFG_TEE_RAM_VA_SIZE, increasing PGT_CACHE_SIZE to be a bit more than half of CFG_TZDRAM_SIZE in MB, and in the trusted application, increasing TA_STACK_SIZE as well as TA_DATA_SIZE [9]. TA_DATA_SIZE is what ultimately determines the heap size of the trusted application.

Aside from the TA heap size, there were also other settings that we changed for this experiment. Table 3 gives a list of all OP-TEE memory-related settings we changed to run this experiment.

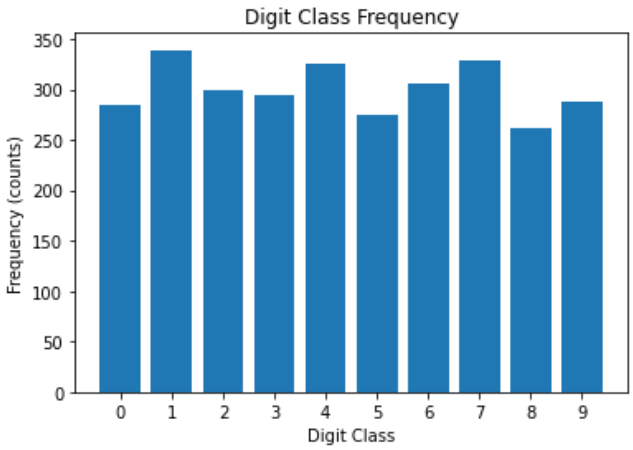

Figure 4: Frequency of each class in the MNIST hand-written dataset used. There are a total of 3000 images.

5.2 Dataset

The dataset we selected for this experiment is the MNIST hand-written digits obtained from DarkneTZ [5]. In total, there are 3000 examples, all of which we used for training. Each example is a 28 x 28 x 1 monochrome image. The pixel

Table 4: FCNN Architecture

Layer	Neurons
Input	784
Hidden layer 1	64
Hidden layer 2	32
Output layer	10

values of the image are rescaled to be between values of 0 and 1. There are a total of 10 classes for these examples, ranging from a digit of 0 to 9. The overall dataset appears to be fairly balanced as shown in Figure 4. The class of digit 8 has the lowest number of samples at 261 while the class of digit 1 has the highest number of samples out of any class at 339. Validation and test sets of the data were not split out from the 3000 samples as we are mostly concerned with the memory and CPU performance of the model in trusted applications; however, it is certainly something worth considering doing in the future.

Due to the overall structure of FCNNs, each 2-dimensional example must be flattened into a 1-dimensional input vector of size 784 x 1. Furthermore, since there are 10 classes, we encoded the labels of every example in one-hot encoding, with the "hot" index corresponding with the value of the digit.

5.3 Network Architecture and Parameters

Our primary focus here is on fully connected neural networks. The particular 3-layer network architecture that we used is summarized by Table 4. The activation function used is the Rectified Linear Unit (ReLU), the loss function is Cross-Entropy loss, our optimizer is Gradient Descent, and we set the learning rate of the model to be a constant 0.03. We varied the mini-batch size to be 50 and 100 samples as well as the number of layers in each partition for layer-based partitioning to be 1, 2, and 3 layers. 3 layers in a partition in our architecture means all layers are in the TA at once; in other words, no layer-based partitioning. Finally, for the sake of time, we only trained the model each time for 10 epochs.

6 RESULTS AND ANALYSIS

In this section, we will first introduce the accuracy results achieved by the aforementioned model on the MNIST dataset in Section 6.1. We will then depict the impact on execution time from running the model under varying conditions such as TEE versus REE, mini-batch size, and layers in a partition in Section 6.2. Finally, we examine the memory performance impacts of the model also under varying mini-batch sizes and layers per partition in Section 6.3.

6.1 Model Accuracy

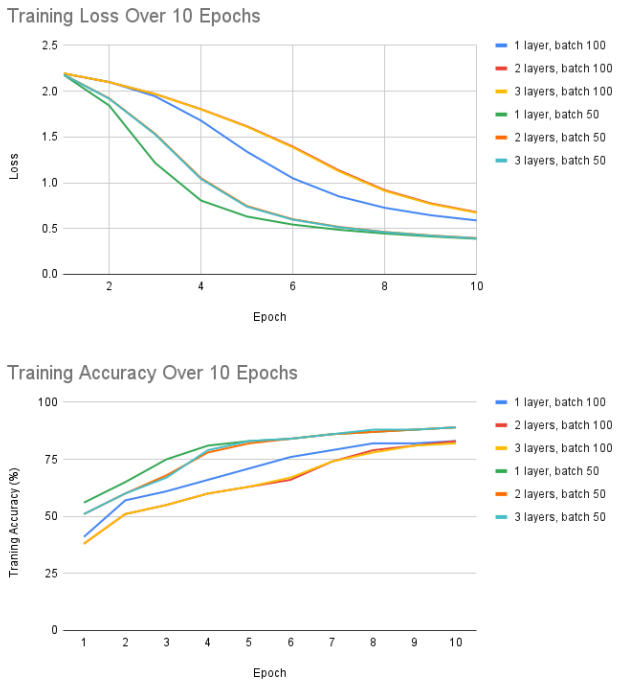

Figure 5: Training loss and accuracy of model over 10 epochs on MNIST for mini-batch sizes of 50 and 100, and 1, 2, and 3 layers per partition.

From our initial test and experimentation, our model was able to get around a 93% training accuracy when trained for 50 epochs with a mini-batch size of 100 samples. However, due to limited time, we had to run most of our experiments with 10 epochs only. Figure 5 gives the training loss and accuracy over 10 epochs for different combinations of batch size and layers loaded.

Overall, it appears that a batch size of 50 results in faster convergence for the model compared to a mini-batch size of 100. In addition, as expected, the training accuracies between the number of layers in a partition for a given mini-batch size are relatively similar. This is because changing layer-based partitioning does not have an effect on the input or parameters of the model.

6.2 Execution Time

The overall execution times of training the model are summarized in Table 5. As depicted, there appears to be around a 10-14% decrease in performance when executing the model in the TEE compared to the REE. This is perhaps due to OP-TEE using the powersave CPU governor in Linux, causing decreased CPU frequency speeds [1]. Additionally, we

Table 5: Execution Time of Training for 10 Epochs

Layers Loaded at Once	Mini-batch Size	Time Elapsed (s)
3 (REE)	100	352.2
3 (REE)	50	365.7
3 (TEE)	100	399.5
3 (TEE)	50	405.4
2 (TEE)	100	8409.7
2 (TEE)	50	12629.0
1 (TEE)	100	5972.2
1 (TEE)	50	8947.3

noticed a very substantial increase in executing time once layer-based partitioning was enabled in the TEE. In fact, the maximum drop in performance we experienced was over 30x compared to executing without layer-based partitioning in the TEE or executing in the REE. This suggests that the overall overhead of swapping layers in and out of permanent secure storage during training is quite substantial.

To further investigate the actual cost of swapping layers in and out of secure storage, Figure 6 plots the distribution of store layer operation times, read layer operation times for when a layer-based partition includes only one layer and the mini-batch size is 100. The time distribution to read read and store layers both appear to be a bimodal distribution. We suspect this distribution to be caused by the fact that the first layer has substantially more weights compared to that of the second and third layers. For example, the weight matrix for layer one is 784×64 with 50,176 values while the weight matrix for layers two and three contain only 2,048 and 320 values respectively.

Another interesting observation is that the overall read layer times are significantly faster than store layer times. We have determined that it does not come from layer serialization and deserialization and instead, comes from the API call to `TEE_CreatePersistentObject` being much slower than `TEE_ReadObjectData`. This is probably because we are using `TEE_CreatePersistentObject` to overwrite existing data. In other words, the `TEE_DATA_FLAG_OVERWRITE` flag ensures that existing objects will get deleted and re-created atomically [2].

Next, comparing the actual time for each batch, Figure 7 plots the distribution of the batch times (the time for one batch to complete) for the model trained without layer-based partitioning as well as with layer-based partition when each partition contains one or two layers - all in the TEE. While the batch times without layer-based partitioning appear to resemble a skew-right distribution with only one peak, both batch times with layer-based partitioning appear bimodal with a few outliers. Regarding the distributions with layer-based partitioning, we discovered that most of the data points in the second, lower peak come from batches in the first 1 to 2 epochs while the first, high peak comes from the other 8 to 9 epochs. We are uncertain of this anomaly and is, therefore, certainly an area of further investigation.

Aside from the shape of the batch times distributions, it is expected to see that the batch times for models run with layer-based partitioning be greater than without the partitioning. What is surprising, however, is batch times of models trained with 2-layer partitions are on average higher than batch times of models trained with only 1-layer partitions. This turns out to be caused by an inefficiency in our layer-partitioning implementation where layers are being swapped in and out of the TA when it is not actually necessary. This therefore also explains the performance penalty between 1 and 2 layers loaded at once as seen in Table 5. If this inefficiency were to be corrected, we suspect the batch time as well as training time to be very similar between the two models run with layer-based partitioning.

While we only showed results for a mini-batch size of 100, results for a mini-batch size of 50 follow a very similar pattern, albeit with lower times (approximately 80% that of mini-batch size 100). However, the reason the total time taken when training the model is greater compared to mini-batch size 100, as seen in table 5, is due to the 100% increase in the number of batches.

6.3 Memory Performance

The initial goal of layer-based partitioning is to reduce the memory footprint of executing DNNs in the TEE. However, its actual implementation brings about a certain level of unforeseen complexity. For example, we had to correct for extra memory taken up in the TA as a result of loading all training examples into the TA at initialization. Its total size is 3000 images * (28 * 28) pixels/image * 8 bytes/pixel, which is approximately 18 MB in total. While we could have loaded the examples in batches during training, this would involve the additional overhead of sending data from the client to the TA via RPC calls.

Table 6 summarizes the actual as well as corrected memory usage of different model configurations including layers loaded in the TA as well as mini-batch size. In general, decreasing batch size results in lower maximum and average memory usage. Things, however, are more complicated when comparing the number of layers loaded into the TA at once. While the overall average memory usage does decrease with the number of layers loaded, the variance does increase with layer-based partitioning, actually resulting in higher maximum utilization as well as lower minimum utilization.

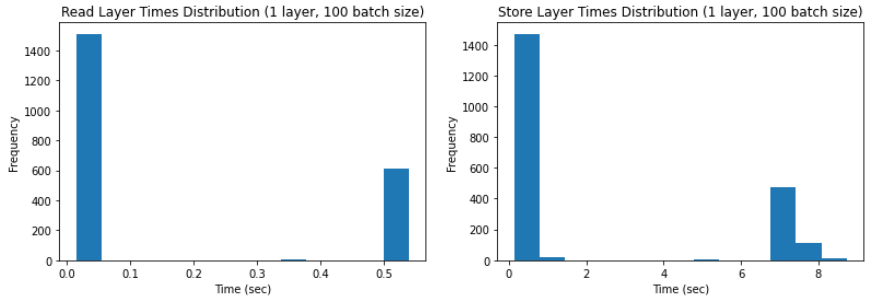

Figure 6: For a layer-based partitioning of a single layer and mini-batch size of 100, shows the (Left) times to read layer and (Right) times to write layer.

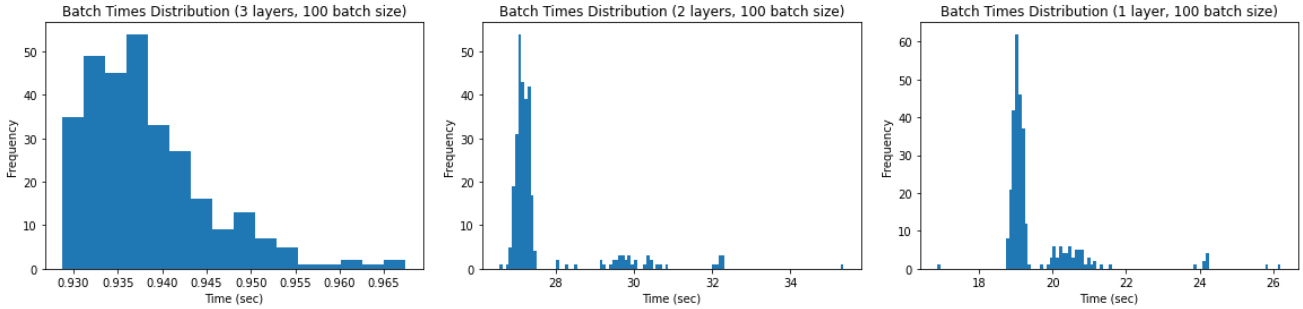

Figure 7: Batch times (Left) without layer-based partitioning in the TEE, (Center) with layer-based partitioning of 2 layers, and (Right) with layer-based partition of a single layer. Mini-batch size of 100 for all 3.

Table 6: Actual and Corrected Memory Usage of FCNN in TEE

Setting		Actual Memory Usage (MB)				Corrected Memory Usage (MB)			
Layers Loaded at Once	Mini-batch Size	μ	σ	min	max	μ	σ	min	max
3	100	20.63	0.29	18.80	21.19	2.69	0.29	0.85	3.25
3	50	19.93	0.14	18.80	20.21	1.98	0.14	0.85	2.26
2	100	19.95	0.98	18.42	22.45	2.00	0.98	0.47	4.51
2	50	19.38	0.74	18.40	21.20	1.44	0.74	0.45	3.25
1	100	19.61	0.91	18.42	22.45	1.67	0.91	0.47	4.51
1	50	19.13	0.68	18.40	21.20	1.19	0.68	0.45	3.25

Figure 8 depicts the average memory utilization of a single batch during training for various layers loaded and mini-batch size configurations. Broadly speaking, it can be seen that these figures match the values in Table 6 and that the variance in memory utilization increases once layer-based partitioning is used. A distinct feature in the plots with layer-based partitioning, also the main culprit of variance in the graph, is the various tall plateaus in the graph that mark the events where layers are being stored and retrieved to and from the secure storage. This is because whenever a layer is stored or retrieved from the secure storage, an array buffer at least the size of the transferred layer will be allocated as a medium to transport data to and from the secure storage and the TA. As a result, this effectively doubles the size of the layers stored or retrieved, bringing about these large, long durations of high TA memory usage.

An obvious method to reduce the memory footprint of these load and store memory operations is to transfer the layer in sections. For example, we can allocate smaller buffers to transfer different fields in the layer one at a time. This, however, will result in the additional time overhead of multiple `TEE_CreatePersistentObject` and `TEE_ReadObjectData` calls to transfer a single layer. Another potential method that builds on top of the previous idea is to store fields in

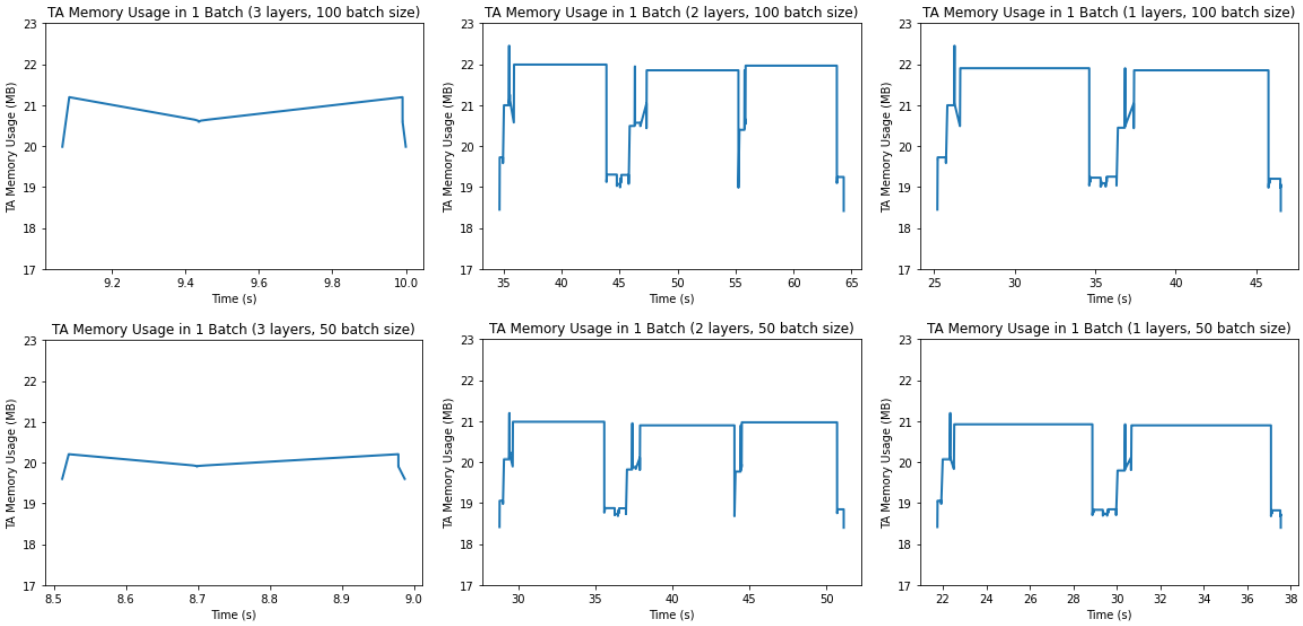

Figure 8: TA memory usage during a single batch. (Top) mini-batch size 100, (Bottom) mini-batch size 50, (Left) without layer-based partitioning, (Center) 2 layers per partition, (Right) 1 layer per partition.

the layer only when necessary or when they are modified; a technique that may also improve execution time since it is the `TEE_CreatePersistentObject` operation that is the main bottleneck. This, however, will require additional information on which fields have been modified. In general, there are many ways to go about this problem and it is certainly something worth considering in the future.

As an aside from memory utilization, the implementation inefficiency as described in Section 6.2 also manifests itself in the center plots with 2 layers loaded. As can be seen, there appear to be extra load and store layer operations compared to only one layer loaded, when there should be less.

Overall, with our current implementation of layer-based partitioning and model configuration, it appears that decreasing the batch size is an even more effective way of reducing the memory footprint when training neural networks in a trusted execution environment. This, however, should be considered with caution as the batch size can affect the performance of the trained model.

7 CONCLUSION

We have implemented layer-based partitioning for securely running DNNs with OP-TEE. We have also demonstrated the time and memory performance of the technique to investigate the utility of such a security approach. Overall, our results indicate that the trade-off between security and performance is unavoidable as it took 10 to 14 percent more time to run an FCNN in the TEE given the same parameters. Aside from the additional performance overhead, we see that layer-based partitioning for the memory-constrained TEE can increase maximum memory usage while offering only marginal improvements to average memory usage, diminishing the utility of the technique. However, there is room for future improvements in performance and memory consumption. Faster implementations for matrix operations can be written to increase temporal performance. Addressing the aforementioned layer-swapping inefficiencies in our implementation can further reduce the additional memory overhead. When considering maximum memory utilization, the main culprit for poor performance is the variance in memory consumption that comes with layer-based partitioning due to buffered data transfers to and from the secure storage. As such, carefully storing only necessary and modified fields in layers can be considered and may even improve run time.

REFERENCES

[1] Julien Amacher and Valerio Schiavoni. 2019. On The Performance of ARM TrustZone. *CoRR* abs/1906.09799 (2019). arXiv:1906.09799 http://arxiv.org/abs/1906.09799

[2] GlobalPlatform. 2021. TEE Client API Specification v1.0: GPD_SPE_007. https://globalplatform.org/specs-library/tee-client-api-specification/

[3] Linaro. [n.d.]. TEE Documentation¶. https://optee.readthedocs.io/en/latest/index.html

[4] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Andrea Cavallaro, and Hamed Haddadi. 2019. Towards Characterizing and Limiting Information Exposure in DNN Layers. *CoRR* abs/1907.06034 (2019).

arXiv:1907.06034 http://arxiv.org/abs/1907.06034

[5] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: Towards Model Privacy at the Edge using Trusted Execution Environments. *CoRR* abs/2004.05703 (2020). arXiv:2004.05703 https://arxiv.org/abs/2004.05703

[6] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[7] Peter M. VanNostrand, Ioannis Kyriazis, Michelle Cheng, Tian Guo, and Robert J. Walls. 2019. Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices. *CoRR* abs/1908.10730 (2019). arXiv:1908.10730 http://arxiv.org/abs/1908.10730

[8] Jeremy Konstantinos Watt, Reza Konstantinos Borhani, and Aggelos Konstantinos Katsaggelos. 2020. *Machine learning refined: foundations, algorithms, and applications.* Cambridge University Press.

[9] xiaxinkai. 2019. RPi3 no enough memory error · Issue 2 · mofanv/darknetz. https://github.com/mofanv/darknetz/issues/2#issuecomment-529763445