Master's Programme in Computer, Communication and Information Sciences

Privacy Preserving Deep Neural Network Prediction using Trusted Hardware

Max Reuter

Aalto University

MASTER'S THESIS

Aalto University
MASTER'S THESIS 2018

Privacy Preserving Deep Neural Network Prediction using Trusted Hardware

Max Reuter

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Otaniemi, Thursday 11th October, 2018

Supervisor:	Professor N. Asokan
Advisor:	Andrew Paverd
Advisor:	Samuel Marchal

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Author	Max Reuter
Title	Privacy Preserving Deep Neural Network Prediction using Trusted Hardware
School	School of Science
Master's programme	Computer, Communication and Information Sciences
Major	Mobile Computing, Services and Security **Code** SCI3045
Supervisor	Professor N. Asokan
Advisor	Andrew Paverd
Advisor	Samuel Marchal
Level	Master's thesis **Date** Thursday 11th October, 2018 **Pages** 78 **Language** English

Abstract

In recent years machine learning has gained a lot of attention not only in the scientific community but also in user-facing applications. Today, many applications utilise machine learning to take advantage of its capabilities. With such applications, users actively or passively input data that is used by state-of-the-art algorithms to generate accurate predictions. Due to the extensive work necessary to fine-tune these algorithms for a specific task, they are predominantly executed in the cloud where they can be protected from competitors or malicious users. As a result, users' privacy might be at risk as their data is sent to and processed by remote cloud services. Depending on the application, users might expose highly sensitive data, meaning a malicious provider could harvest extensive amounts of personal data from its users.

In order to protect user privacy without compromising the confidentiality guarantees of traditional solutions, we propose using trusted hardware for privacy preserving deep neural network predictions. Our solution consists of a hardware-backed prediction service and a client device that connects to said service. All machine learning computations executed by the prediction service that depend on input data are protected by a trusted hardware component, called a Trusted Execution Environment. This can be verified by users via remote attestation to ensure their data remains protected. In addition, we have built a proof-of-concept implementation of our solution using Intel Software Guard Extensions (SGX). Compared to existing solutions relying on homomorphic encryption, our proof-of-concept implementation vastly increases the set of supported machine learning algorithms. Moreover, our implementation is tightly integrated into the existing pipeline of machine learning tools by supporting the Open Neural Network Exchange (ONNX) Format. Furthermore, we focus on minimising our Trusted Computing Base (TCB), thus our proof-of-concept implementation only consists of $4,500$ lines of code. Additionally, we achieve a $7\times$ increase in throughput whilst decreasing the latency $40\times$ compared to prior work. In our tests, SGX reduced throughput by 11% and increased latency by 21% compared to our baseline implementation without SGX.

Keywords Machine Learning, Platform Security, Privacy, Trusted Hardware

Contents

1. Introduction

In recent years data has become increasingly important to the point where data can be seen as an asset just like stocks or patents. This has lead companies to collect more data than ever, especially with the rise of machine learning that enables companies to use their aggregated data sets. Machine learning provides a powerful toolset that allows one to make predictions for given data based on previous observations. Due to major advances in the scientific community, we can successfully utilise machine learning techniques to efficiently solve complex problems that seemed impossible to tackle just years before. Consequently, increasingly more developers leverage the capabilities of machine learning to build sophisticated, intelligent applications that work on their users' data. The data that is being operated on ranges from financial data to music or images depending on the application. When talking about data in the context of machine learning, however, one must differentiate between *model data* and *input data* as well as *output data*:

Model data describes the machine learning algorithms and their parameters that are used to compute a prediction for a given input. Together, the used algorithms and their parameters form a *machine learning model*.

Input data represents the inputs that a user sends to a model.

Output data defines the predictions a model generates based on the received input.

There are two prevailing approaches to offer machine learning services: *cloud-assisted* and *on-device* machine learning. Cloud-assisted machine learning describes deploying a machine learning model online so that it accepts requests through a well defined, public interface. This model serves as an oracle which accepts queries and responds with predictions. On-device machine learning, on the contrary, stores the model on the user device itself, meaning no online connectivity is needed to receive predictions. This only has become a possibility in recent years as user devices have become sufficiently powerful for this type of task.

1.1 Problem Overview

Whenever one talks about data, the question of ownership and protection arises which is also the case in these circumstances. Two parties are involved in a typical machine learning scenario:

Service providers own a machine learning model that they want to use for a prediction service.

Users wish to use said prediction service by providing their own, potentially sensitive, data.

Each of these parties wants to protect their data whilst still playing their respective role in this scenario. As mentioned before, a popular way of providing

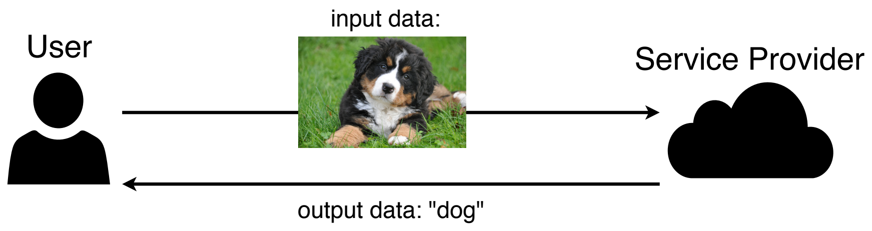

Figure 1.1. Sending user input to a service provider to receive a prediction

a machine learning service is to make the service publicly available through an interface so that it is able to answer queries that are sent to it, as figure 1.1 illustrates. The reason for this approach is that well trained models are very valuable to service providers and therefore they want to protect their intellectual property. As an example, the video streaming service Netflix estimates that its personalisation and recommendation engines save it more than one billion dollars per year [14].

Sending private data to remote servers is troublesome to users, as the data is out of their control where they can no longer protect it. The alternative way of storing models on user devices is worrisome to service providers, however, as they are not willing to hand their valuable model data into the hands of potentially malicious users or competitors.

A proposed approach to address theses issues is to use homomorphic encryption. Homomorphic encryption schemes allow one to perform certain operations on encrypted data without ever decrypting it [12]. This enables service providers to compute predictions on encrypted data. Thus, the content of any input data is never revealed. Previous work using different versions of homomorphic encryption, however, has shown that encryption schemes add a large performance overhead. Moreover, depending on the encryption scheme, only a limited set of algorithms may be used in the deployed model or certain calculations can only be approximated rather than precisely calculated [13][35]. As machine learning models become more sophisticated and are deployed in scenarios where latency is crucial, we must find solutions that address these drawbacks.

1.2 Proposed Approach

One way to address performance and design limitations is to use trusted hardware. By performing computations inside a *Trusted Execution Environment* (TEE), one can operate on cleartext data without revealing the contents to the outside. This could be utilised to deploy a machine learning model on a TEE-enabled server and compute predictions for given inputs inside the TEE. As a result, a service provider could not learn the contents of input and output data while still being in control of his model data. Since the actual computations would be performed on clear text data, neither would the design of our model be limited, nor would our performance be impacted as it would be by using homomorphic encryption. In addition, one could utilise the TEE on the server to attest to a user device that all computations will be performed inside a TEE, meaning that all input data is protected. This way a user device could verify that the server is using a TEE for computing predictions before sending any actual input data.

1.3 Research Scope and Goals

The main goal of this thesis is to investigate the use of trusted hardware for privacy preserving machine learning services and to compare it to existing solutions that rely entirely on encryption. For this comparison, we focus on functionality, security, and performance. Thus, we want to answer the following questions that our approach raises:

1. What functional limitations does trusted hardware impose on cloud-assisted machine learning?

2. How does the performance of our approach compare to existing privacy preserving solutions and to state-of-the-art frameworks which do not preserve privacy?

3. How does our approach compare to existing privacy preserving solutions in terms of security and privacy?

In addition, we consider the following aspects to be beyond the scope of this thesis:

- Security of user devices
- Adversarial server that is trying to steal the model it is hosting

1.4 Research Approach

In order to answer the research questions we defined in section 1.3, we first propose a general system design for cloud-assisted machine learning that relies on trusted hardware. Afterwards, we present a proof-of-concept implementation that uses *Intel Software Guard Extensions* (SGX) as TEE. Finally, we use our implementation to evaluate our approach and compare it with existing solutions as well as state-of-the-art frameworks. To reiterate, this thesis makes the following contributions:

1. **Novel system design** for privacy preserving machine learning
2. **SGX-compatible C++ library** that can interpret arbitrary ONNX models
3. **Evaluation** comparing our implementation to state-of-the-art frameworks, as well as other privacy preserving solutions

2. Background

2.1 Machine Learning

Machine Learning describes the concept of an algorithm being able to learn from raw data. A common definition of machine learning provided by Tom Mitchell in 1977 reads [15]:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

The majority of machine learning algorithms can be categorised into supervised and unsupervised machine learning. For this thesis, we only focus on the former. Supervised machine learning generally defines a function $\mathcal{F}$ that maps values of an input space X to an output space Y [15]. Usually, we want the size of Y to be significantly smaller than the size of X, meaning that $\mathcal{F}$ is able to abstract elements $x \in X$ to simpler elements $y \in Y$:

$$\mathcal{F}(x) = y \tag{2.1}$$

2.1.1 Deep Neural Networks

In general, a neural network is one way to define the function $\mathcal{F}$ described in equation 2.1. Using neural networks, $\mathcal{F}$ can take various forms and usually involves certain parameters that are learned during a training phase. For instance,

$$\mathcal{F}(x) = \Theta_1 * x + \Theta_0 \tag{2.2}$$

forms a simple neural network. Here, Θ denotes the *parameters* of $\mathcal{F}$, with Θ_1 being commonly being referred to as *weight* and Θ_0 as *bias*. Complex neural networks usually describe a composite function, meaning $\mathcal{F}$ is comprised of many different functions f_1, f_2,..., f_n that are chained together:

$$\mathcal{F}(x) = f_n(f_{n-1}(f_{n-2}(...f_1(x)...))) \tag{2.3}$$

Moreover, each function f_j defines its own set of parameters $\Theta^{(j)}$ and has the same form as shown in equation 2.2. Figure 2.1 further illustrates how a network $\mathcal{F}$ is comprised. In this example, f_1 describes the *first layer* of $\mathcal{F}$, f_2 describes the *second layer*, and f_n describes the n^{th} *layer*. Therefore, we can say $\mathcal{F}$ has n layers, or $\mathcal{F}$ has *depth* n. Thus, $\mathcal{F}$ depicts a *deep neural network* (DNN) [15].

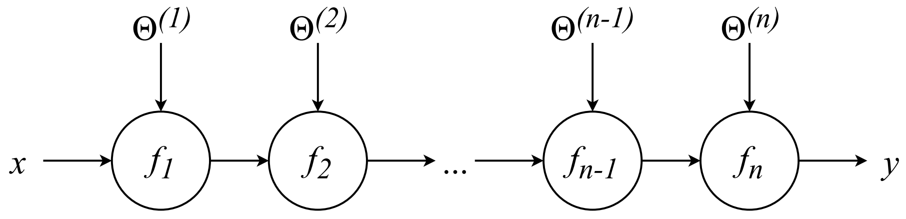

Figure 2.1. Conceptual illustration of a network $\mathcal{F}$ with n layers

2.1.2 Model Representation

Generally, a DNN machine learning model has the following form:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} \; \end{bmatrix} \rightarrow h_\theta(x) \tag{2.4}$$

Here, we input a 3-dimensional vector into our first layer, called *input layer*. Afterwards, we forward our input through 0 to n intermediate layers, called *hidden layers*. Finally, our processed input reaches the last layer, or *output layer*, of our model which outputs the final prediction of our *hypothesis function*, called h_θ [15]. As our input is multidimensional, each layer defines multiple nodes, that each accept a single value as input. We call the nodes *activation units* and label them as $a_i^{(j)}$, where i denotes the index of a node of layer j. Moreover, our model defines a parameter set θ, where $\Theta^{(j)} \in \theta$ describes the parameter matrix used for layer j. With one hidden layer, our model could look as follows:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_\theta(x) \tag{2.5}$$

In addition, each layer also defines a function g that is applied to all its nodes. We call this function *activation function*. Typically, these functions involve some simple mathematical operations. We will cover activation functions in more detail in section 2.1.3.
Calculating the output of our example model could look like the following:

$$\begin{aligned} a_1^{(2)} &= g\big(\Theta_{10}^{(1)}x_1 + \Theta_{11}^{(1)}x_2 + \Theta_{12}^{(1)}x_3\big) \\ a_2^{(2)} &= g\big(\Theta_{20}^{(1)}x_1 + \Theta_{21}^{(1)}x_2 + \Theta_{22}^{(1)}x_3\big) \\ a_3^{(2)} &= g\big(\Theta_{30}^{(1)}x_1 + \Theta_{31}^{(1)}x_2 + \Theta_{32}^{(1)}x_3\big) \\ h_\theta(x) = a_1^{(3)} &= g\big(\Theta_{10}^{(2)}a_1^{(2)} + \Theta_{11}^{(2)}a_2^{(2)} + \Theta_{12}^{(2)}a_3^{(2)}\big) \end{aligned} \tag{2.6}$$

In this example θ contains two parameter matrices $\Theta^{(1)}$ (3×3) and $\Theta^{(2)}$ (1×3) for its two layers. Furthermore, we can see how the output for our second layer is fed to our third layer as input, which ultimately computes the overall output.

2.1.3 Neural Network Operations

In the previous section, each layer of our sample DNN used the same operation to calculate its output. This is usually not the case, as different operations are combined to achieve the desired output. Although there are many different types of layers that each compute a particular operation, for this thesis it suffices to understand the following layers which are among the most common layer types [15]:

Fully Connected Layer

A *Fully Connected Layer* describes the layer we saw above in equation 2.2 where we have full connections to all activation functions of the previous layer. Hence, this layer consists of a matrix multiplication with added bias values, followed by element-wise applying our activation function.

Convolutional Layer

The *Convolutional Layer*, as proposed by LeCun *et al.* [33], is often used in networks that accept images as input. As a result, inputs are usually 3-dimensional. The parameters of a convolutional layer consist of many small, usually squared, filters that are applied to the input. During computation, each filter is moved across the input to compute the dot product between the values of the filter and the values of the input at the current position. For example, if our input has dimensions $3 \times 30 \times 30$ and our filters are of size $3 \times 5 \times 5$, each filter produces a 2-dimensional output, or *activation map*. Assuming we have 10 filters, we compute 10 different activation maps, which, when stacked, produce the output of the

Convolutional Layer. Figure 2.2 illustrates how a filter is used to compute an activation map.

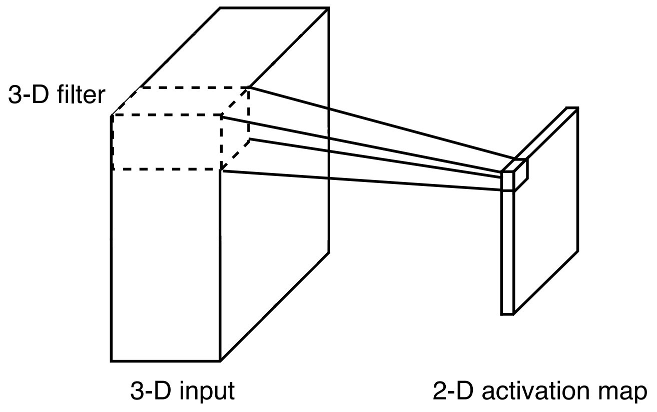

Figure 2.2. Illustration of moving a single filter across the input to compute an activation map

MaxPool Layer

The *MaxPool Layer* is often used in combination with a Convolutional Layer. Similar to a Convolutional Layer, the MaxPool Layer moves a, usually squared, filter across its input. However, instead of computing a dot product, the filter is used to find the maximum value of the input at the current position. Only the maximum value at each position is kept and all other values are discarded. An example of how the MaxPool Layer computes its output is depicted by figure 2.3. The MaxPool operation is done independently for each depth slice. As a result, the MaxPool Layer reduces the height and width of its input but leaves the depth unchanged.

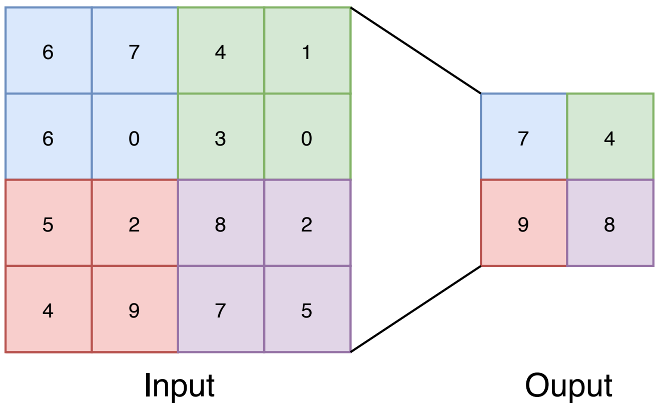

Figure 2.3. Example of reducing the spatial height and width of an input using MaxPool with each colour denoting the input and output of a filter

Activation Functions

Just like there are different types of layers, there are also different types of activation functions. Three common activation functions that we use for this thesis are [15]:

ReLU. The activation function *Rectified Linear Unit* (ReLU) is used to discard negative values. Its mathematical definition is:

$$ReLU(x) = max(0, x) \tag{2.7}$$

Sigmoid. The *Sigmoid* activation function allows mapping values to the interval $(0, 1)$ and is defined as follows:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

TanH. Similar to the Sigmoid activation function, *TanH* allows mapping input values to a certain range. For TanH, this range is $(-1, 1)$. The mathematical definition of TanH is:

$$TanH(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.9}$$

2.1.4 Open Neural Network Exchange

The *Open Neural Network Exchange*[1] (ONNX) format is an open source project developed by Facebook and Microsoft. The goal of ONNX is to create an interoperable format for representing neural networks that is compatible with different machine learning tools, such as frameworks and runtimes. As of writing this thesis (October 2018), all major machine learning frameworks, including Caffe2, PyTorch, TensorFlow[2], and SciKit-Learn[3], support exporting a model to ONNX format.

For representing a model, ONNX defines a top-level *Model* component. The purpose of this component is to store metadata associative to the machine learning model it describes, e.g. the model producer, as well as a *Graph* component. ONNX drops the notion of layers and represents all atomic operations of a model, including activation functions, as individual nodes of this graph component [42]. For the scope of this thesis, it suffices to discuss only the most general components. For the rest of this section we explain said components, starting from the ground up [42]:

Tensor

The *Tensor* component describes values passing through the graph from one operation to another. Generally, a Tensor component defines a name, a data type, a shape, and a data field. The name of a Tensor uniquely identifies it across

[1]https://onnx.ai/
[2]https://www.tensorflow.org/
[3]http://scikit-learn.org/stable/

the Graph component it belongs to. The data type specifies what data is held by the Tensor. ONNX defines multiple data types that are supported by Tensors, ranging from basic types such as `int` or `float` to advanced type such as complex numbers. The data field contains the data held by the Tensor. Multiple data fields are defined to allow storing different data types, but only one is ever used. Which field is used, is defined by the data type of a Tensor. Lastly, the shape property describes how the held data is organised, i.e. if the Tensor stores a scalar, a vector, a matrix, or something of higher dimensionality.

ValueInfo

The *ValueInfo* component defines basic information about a value, which usually refers to a Tensor. Therefore, a ValueInfo defines all properties that describe a Tensor, i.e. its name, data type, and shape. As a result, a ValueInfo component can be used to describe what kind of Tensor is expected, for instance as an input.

Attribute

The *Attribute* component represents a container for a runtime constant, which can have various data types. In order to be able to support storing different data types, the Attribute component defines, similar to the Tensor component, multiple data fields ranging from a single `float` or `int` over a list of `float` or `int` values to a string. However, only one of these fields actually holds data. Which field that is, is determined by the data type property. Moreover, an Attribute also defines a name that describes the purpose of the Attribute. For instance, if we want to multiply two matrices A and B, we can use two Attribute components, one for A and one for B, to tell us whether the matrices should be transposed before they are multiplied with each other. As opposed to other components, the name of an Attribute is not unique.

Node

The *Node* component describes a single node inside the Graph and is comprised of a list of inputs as well as outputs, a list of attributes, and its operator type. Input and outputs are stored in two separate lists of strings. These strings refer to names of Tensor components, meaning a node defines what Tensors it is expecting as inputs and what Tensors it outputs. It is sufficient, to only store the name of each Tensor, as their names must be unique across the Graph. Each Node component represents a machine learning operation. The kind of operation a Node portrays is defined by its operator type field. Finally, the list of Attributes stored by a Node describe the runtime constants needed to execute the operation a Node defines.

Graph

As mentioned before, the *Graph* component contains all operations defined by a machine learning model and stores them in Node components. Besides a list of Nodes, the Graph component also contains a list of initialisers and the expected input as well as output. Input and output of a Graph are stored as ValueInfo components, meaning everything but the actual data is pre-defined. Finally, the list of initialisers describes a list of constant Tensor components that are needed by different Nodes. This list of Tensors represents θ, meaning the parameter set of the model, as described in section 2.1.2.

All components highlighted above and their relationships between each other are illustrated by figure 2.4.

2.2 Trusted Execution Environment

The concept of a Trusted Execution Environment (TEE) in general describes an isolated execution environment that runs alongside a Rich Execution Environment (REE). Whilst the operating system and ordinary, *untrusted applications* are executed in the REE, the TEE provides a safe execution environment for authorised, *trusted applications*. In addition, the TEE ensures data integrity and confidentiality as well as enforces access control to resources belonging to trusted applications. As a result, all trusted applications are independent from one another and cannot access each others' resources or data. Moreover, the TEE might enable access to internal resources, such as specialised hardware for cryptographic operations, to trusted applications. Figure 2.5 illustrates a possible architecture for a generic, hardware-based TEE, as described by Asokan *et al.* [3]. A TEE provides a well-defined interface for untrusted and trusted applications to communicate. Therefore, trusted applications can offer services to untrusted applications. Consequently, developers may choose to split their application into a trusted and an untrusted component which are tightly interconnected. This allows applications to outsource sensitive operations, e.g. to securely compute cryptographic functions, to their trusted component which is executed inside the TEE. For our solution, we rely on the following properties of a TEE:

Isolated Execution: code is executed in an isolated environment that is inaccessible to any unauthorised application, trusted or untrusted

Remote Attestation: a trusted application is able to attest to a remote party that it is executed inside a TEE

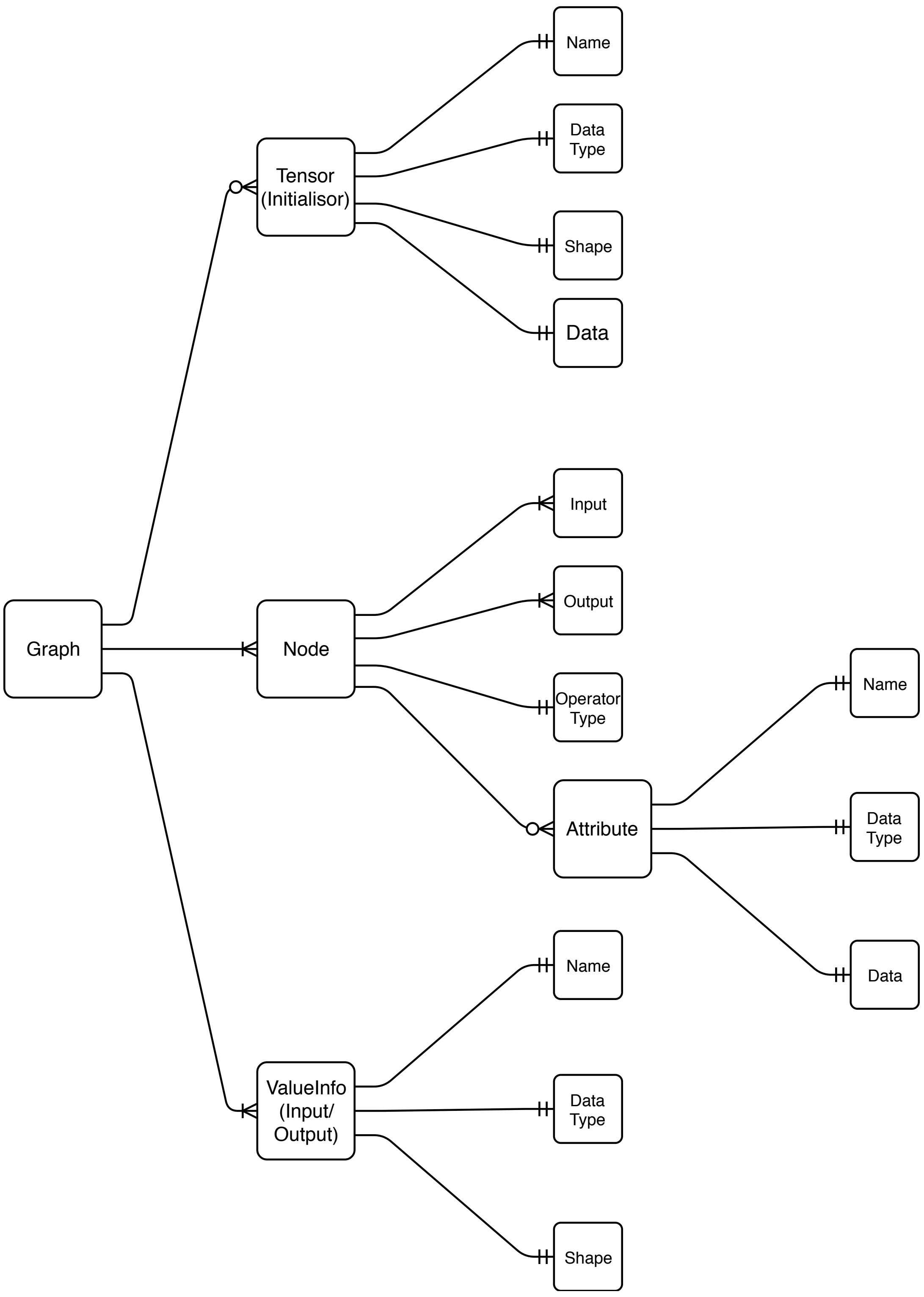

Figure 2.4. Overview of highlighted ONNX components

Due to code being executed in an isolated environment, the code and its data are protected against tampering as well as unauthorised access. Remote attestation, on the other hand, enables a trusted application to establish trust with another application that runs on a different device.

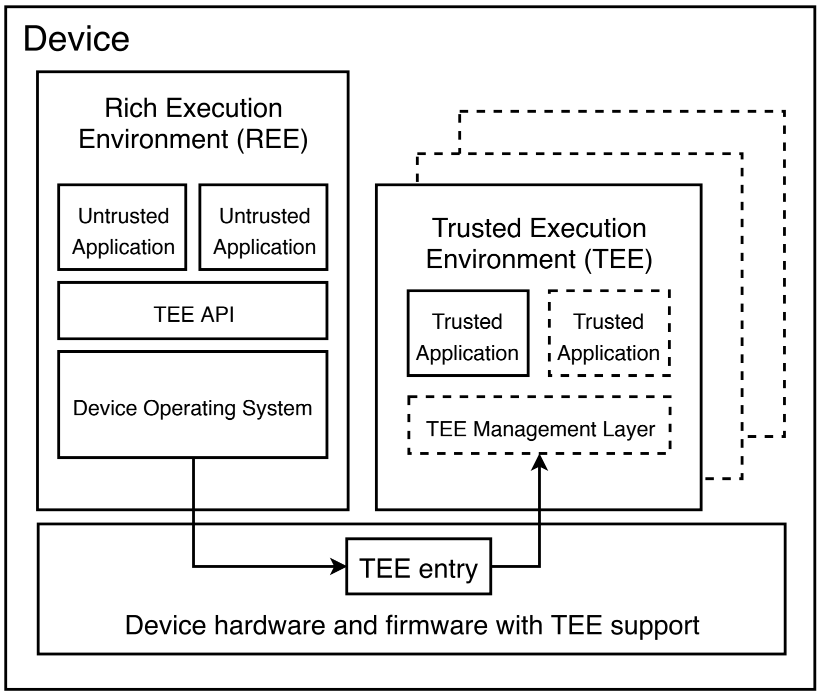

Figure 2.5. Architecture of a generic TEE [3]

2.2.1 Remote Attestation

As mentioned before, remote attestation allows one party, i.e. the *prover* to attest to another, remote party, i.e. the *verifier*, that it has sufficient integrity to be trusted. In order to achieve this, remote attestation has to provide the following mechanisms [47]:

Measurement Mechanism: allow a prover to determine what components of the execution environment need to be measured, when they need to be measured, and how to securely store measurements

Integrity Challenge Mechanism: enable a verifier to obtain a list of measurements of an attesting system and verify its completeness and freshness

Integrity Validation Mechanism: enable a verifier to validate a list of measurements of an attesting system by verifying the list is complete, non-tampered, fresh, and that all individual measurements describe a trustworthy execution environment

To attest its trustworthiness, the TEE hardware of a prover computes a *quote*, which contains the measurement of the prover. In addition, a quote might also include a challenge, e.g. a random number, supplied by the verifier to ensure freshness of the quote. Moreover, the quote must be authenticated, i.e. signed, by the TEE hardware to ensure the quote was generated by a valid TEE. Afterwards, the prover sends the quote to the verifier who checks the following:

1. the signature of the quote to ensure its authenticity
2. whether the quote contains the correct challenge to ensure its freshness
3. the measurement of the prover to ensure the correct software is executed

As remote attestation requires a full round trip, it can also be used to establish a shared key between verifier and prover, e.g. by using Diffie-Hellman key exchange. Thus, once remote attestation was completed successfully, both parties have established a secure communication channel.

2.2.2 Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) describes a set of new CPU instructions as well as memory access changes introduced with Intel's Skylake architecture [18][36]. As an implementation of a TEE, SGX enables untrusted applications to instantiate a protected container, called *enclave*. The code that is executed by an enclave is known in advance, meaning it can be analysed by any party who has access to it. Moreover, as an enclave is starting up, the SGX hardware measures the enclave's code and data. This measurement can later be used for remote attestation. After an enclave has been created and all its content has been loaded as well as measured, SGX enforces strict access control. Hence, all resources and data of any enclave are protected against any external access, including access attempts by privileged processes [36].

SGX Hardware Components

Intel SGX includes different hardware components that are needed to protect enclave resources. Memory used by enclaves are stored on *enclave pages* which are, among other SGX structures, maintained inside the *enclave page cache* (EPC). With current (October 2018) SGX hardware, the EPC can allocate about 90 MB for all enclaves combined. If more memory is needed, SGX introduces paging to oversubscribe EPC memory [27]. Whenever software tries to access an enclave page, SGX utilises the *enclave page cache map* (EPCM) to determine to which enclave the requested page belongs and whether to grant access [36]. To cryptographically protect data stored in the EPC, SGX employs a *memory encryption engine* (MEE). Due to the MEE, enclave pages are encrypted when stored in main memory and decrypted as they are loaded into the CPU, making contents available in plaintext [18].

Enclave Life Cycle

As mentioned above, SGX enables applications to launch an enclave and communicate with it. In addition, an application is responsible to destroy each enclave it creates. Figure 2.6 depicts the typical life cycle of an enclave. Each enclave

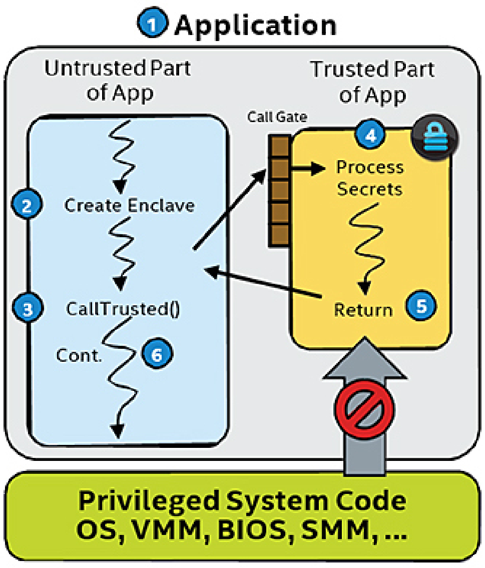

Figure 2.6. Life cycle of an SGX enclave [23]

and corresponding untrusted application define an interface over which they can communicate with each other. An untrusted application can call functions made available by the enclave via an *enclave call* (ECALL). Enclaves, on the other side, can call functions made available by untrusted applications via an *outside call* (OCALL) [25]. This allows both trusted and untrusted components of an application to delegate certain operations to one another, depending its requirements.

Remote Attestation in SGX

Intel SGX supports remote attestation, allowing a user-created enclave to attest to remote parties that it has been correctly initialised and that all data is protected by SGX. As mentioned before, the enclave measures the contents of its enclave pages as it is built. This measurement is stored in *MRENCLAVE*, a SHA-256 digest that represents the identity of an enclave. This measurement is only taken once when the enclave starts, after which the enclave is locked an cannot load any more code. Therefore, any data, such as potential inputs that are provided to the enclave via ECALLs, is not reflected by the value of MRENCLAVE [2].
SGX implements remote attestation by involving a local enclave called *Quoting Enclave*. The Quoting Enclave is provided by Intel and will attest to any remote verifier the integrity of any user enclave on behalf of that enclave. For the Quoting enclave to attest this, a user enclave must perform local attestation to prove its integrity to the Quoting Enclave. Local attestation between a user enclave and the Quoting Enclave works by the user enclave generating a *REPORT*. This REPORT structure contains attributes of the enclave, the trustworthiness of the hardware, optional user data, and a MAC tag. The MAC is produced by the CPU using

AES128-CMAC over the remaining REPORT data with the *Report Key*. Given a REPORT, the Quoting Enclave, can verify the integrity of the corresponding enclave. If the REPORT is valid, the Quoting Enclave replaces the MAC tag of the REPORT with a signature using a device specific *Intel Enhanced Privacy ID* (EPID) key. This new structure is called a *QUOTE*. Finally, when a verifier receives a QUOTE, he can check whether it stems from a valid Quoting Enclave and therefore from a trustworthy user enclave by consulting *Intel Attestation Services* (IAS). As Quoting Enclaves are provided by Intel, only Intel can verify if a QUOTE is valid [2].

2.3 Previous Work

As mentioned in section 1.1, previous work has addressed privacy preserving machine learning before using homomorphic encryption. For this thesis, we compare our work to the following encryption-based solutions:

CryptoNets

Dowlin *et al.* [13] describe CryptoNets, i.e. privacy preserving neural networks that operate on encrypted data. CryptoNets relies on *levelled homomorphic encryption*, a weaker variant of fully homomorphic encryption. Therefore, CryptoNets does not support arbitrary operations on encrypted data which has to be taken into account when designing a model or planning on reusing an existing model. For their empirical tests, Dowlin *et al.* trained a model on the MNIST[4] data set. After the training phase, they had to simplify the resulting model slightly due to the limitations of the encryption scheme of CryptoNets. Nonetheless, the changes made did not affect the achieved accuracy of 99%. CryptoNets is optimised for throughput, meaning it can process thousands of images in parallel. As a result, CryptoNets, achieves $58,982$ predictions per hour during the tests. Thus, CryptoNets scales reasonable well, if a single user submits batches of thousands of images at a time. However, CryptoNets has poor latency, meaning that processing a single image takes an extensive period of time. This is due to its design and encryption scheme that always compute on the entire possible input space, regardless of how much is actually used. Hence, processing 1 image or a batch of 4096 images requires the computation time of 297.5 seconds. Moreover, the messages sizes required to send inputs to CryptoNets are considerable. For instance, sending 4096 encrypted greyscale images with 28×28 pixels as input results in a total message size of 367.5 MB or 91.875 KB per image.

[4]http://yann.lecun.com/exdb/mnist/

MiniONN

MiniONN was developed by Liu *et al.* [35] and describes a transformation technique that enables converting common neural network models to oblivious neural networks (ONN). MiniONN introduces oblivious protocols for operations frequently used in neural networks. As a result, whenever a client sends an input to the server, both parties exchange messages for every layer of the ONN deployed on the server. In addition to oblivious protocols, MiniONN also relies on *additively homomorphic encryption* to further ensure model parameters and input values remain private to the appropriate party. To increase performance, MiniONN only uses lightweight cryptographic operations during the online prediction phase and allows precomputing input-independent values in an offline phase. In their evaluation, Liu *et al.* report that they vastly outperform CryptoNets in terms of message sizes and latency. Utilising the same, simplified network trained on the MNIST data set as CryptoNets, MiniONN reduces latency to 0.4 seconds and message sizes to 44 MB, whilst maintaining the accuracy reported by CryptoNets. In addition, Liu *et al.* note that MiniONN enables model designers to trade off prediction accuracy to reduce the overhead of the resulting ONN. Moreover, MiniONN supports all common machine learning operations and therefore offers a significantly more extensive operator set to model designers than CryptoNets. However, due to its encryption scheme, smooth activation functions, such as TanH and Sigmoid, can only be approximated. Furthermore, MiniONN only protects the values of model parameters and inputs. In fact, due to its design, MiniONN reveals the sizes of inputs, sizes of parameters matrices, and the structure of the model to client.

2.4 Gap Analysis

Even though CryptoNets achieves privacy preserving machine learning, it introduces large drawbacks, such as poor latency and a large overhead for message sizes. Furthermore, CryptoNets can only support a limited set of machine learning operations. MiniONN improves on the drawbacks of CryptoNets, but introduces other disadvantages, such as revealing the model structure to the client. In addition, although better than CryptoNets, MiniONN's performance is still worse than what non privacy preserving frameworks offer. Lastly, neither CryptoNets nor MiniONN are able to analyse inputs for possible filtering or metering on the server side as inputs are encrypted.

3. Adversary Model and Requirements

In this chapter we define and motivate our adversary model. In addition, we also outline what requirements our system has to meet as well as what goals we set for ourselves.

3.1 Problem Statement

Personal data, i.e. users' input and output data of a machine learning model, is a valuable asset. It can be turned into financial profit in a multitude of ways, such as by improving an already existing machine learning model to gain a business advantage over competitors or by simply selling the data to a third party.
Knowledge about model data, on the other hand, can be exploited in different ways. For instance, this information can be used to improve competing models, for example by adapting the same architecture. Tramèr *et al.* [52] have shown that extensive knowledge about a remote model can even be exploited to rebuild a copy of the model, allowing a malicious user to compute predictions without querying the service provider. Alternatively, Papernot *et al.* [44] have demonstrated that acquired insight of a deployed model can be used to generate *adversarial samples*. These samples are specifically crafted to be mispredicted by a model, potentially endangering the security guarantees of a deployed model or causing financial harm.

3.2 Adversary Model

For this thesis, we consider three different types of adversaries: malicious service provider, malicious user, and malicious outsider. Each type of adversary has different objectives and capabilities, which we will cover in the following sections.

3.2.1 Malicious Service Provider

A malicious service provider has full authority over both software and hardware of his server. However, we assume that the used hardware is not compromised and works as intended by the manufacturer. More precisely, we assume the TEE hardware of the server is not compromised, thus providing an isolated execution environment and remote attestation capabilities for any code that is executed inside. Consequently, all input-dependent machine learning computations can be performed inside the TEE without leaking secrets to the outside. As a result, a malicious service provider can monitor, inspect, and edit any data arriving at the server, including user queries and their metadata. Moreover, he can decide what model architecture and parameter values are being used to compute predictions for incoming queries.

The main drive of a malicious service provider is to collect as much data about his users as possible. The data that is of interest in our scenario is input data sent to the model, which can range from sensor data to images, and its computed output data, i.e. the prediction. In addition, metadata such as location and timestamps might also be valuable. A malicious service provider is malicious from the point of view of a user, i.e. a malicious service provider does not aim to steal models of other service providers.

3.2.2 Malicious User

We consider a malicious user to be a legitimate user with malicious intent. Moreover, we assume that a malicious user is not able to compromise the server of a service provider. Therefore, a malicious user has full control over what input data she provides. This enables her to craft inputs, also based on previously seen output data.

Similar to a malicious service provider, the major objective of a malicious user is to collect data about the service provider and its service. Primarily, a malicious user wants to gather model data.

3.2.3 Malicious Outsider

A malicious outsider has neither access to user devices or the server of a service provider, but the network in-between. Consequently, a malicious outsider can inspect, edit, delay, and drop any network packets sent between user and service provider. Furthermore, he is able to act as a user, allowing him to craft his own input data and receive the computed predictions.

As a third party, a malicious outsider combines the goals of malicious service provider and malicious user with the ambition to obtain both model data and input as well as output data.

3.3 Solution Requirements

We divide our requirements into security and functional requirements. Each group covers different aspects of the system and allows us to easily compare our approach with other existing solutions.

3.3.1 Security Requirements

The security requirements form the foundation of our approach and therefore have the highest priority. We define our security requirements as follows:

S-1 **Model confidentiality:** Model data must stay secret and no information regarding the model must leak outside the server.

S-2 **User privacy:** Input and output data must remain private to their respective user.

3.3.2 Functional Requirements

Even a perfectly secure and privacy-preserving approach is of no use if it is not applicable for any real world system. Thus, our functional requirements are of the second highest priority. The exact requirements are listed below:

F-1 **Accuracy:** The accuracy of a model must not decrease when using our approach. Hence, for any given model $\mathcal{M}$ that achieves accuracy $\mathcal{A}$ using a state-of-the-art machine learning framework and accuracy $\mathcal{A}'$ when using an ideal implementation of our approach, $A = A'$ must hold.

F-2 **Coverage:** Our approach must not affect the process of designing a new model. Thus, let $\mathcal{O}$ denote the set of popular operators, i.e. machine learning algorithms, when using a state-of-the-art framework and let $\mathcal{O}'$ denote the set of operators supported by our approach. Then $\mathcal{O}' = \mathcal{O}$ must hold. Furthermore, as $\mathcal{O}$ grows in size over time as more algorithms are developed in the future, our approach must allow to easily extend $\mathcal{O}'$ to ensure $\mathcal{O}' = \mathcal{O}$ still holds. In addition, our approach must not limit the size $\mathcal{S}$, measured in bytes, of any model $\mathcal{M}$.

F-3 **Non-invasiveness:** Given an existing model $\mathcal{M}$, our approach must support

$\mathcal{M}$ without modifications with the same guarantees as when designing a new model.

3.4 Solution Goals

In addition to our requirements, which our solution must meet, we also define a set of goals we pursue for our approach.

3.4.1 Performance Goals

In order to compete with existing state-of-the-art machine learning frameworks, our solution should provide comparable performance. Our goal is that the overhead introduced by our approach is a constant factor for both overall throughput and latency. The detailed goals are the following:

P-1 **Throughput:** Assume a state-of-the-art framework achieves a throughput of $\mathcal{T}$, measured in queries per second, for a given model $\mathcal{M}$. Then an ideal implementation of our approach, given model $\mathcal{M}$, should achieve a throughput $\mathcal{T}'$ such that $\mathcal{T}' \leq c \cdot \mathcal{T}$ holds with c being a constant.

P-2 **Latency:** Let $\mathcal{L}$ denote the latency for computing the prediction for a given input $\mathcal{I}$ and model $\mathcal{M}$ when using a state-of-the-art framework. Then an ideal implementation of our approach, given $\mathcal{I}$ and $\mathcal{M}$, should achieve latency $\mathcal{L}'$ such that $\mathcal{L}' \leq c \cdot \mathcal{L}$ holds with c being a constant.

3.4.2 Deployability Goals

To incentivise a maximum adoption rate of new systems, they should aim to meet certain deployability goals. In our case, the goals are as follows:

D-1 **Commodity server hardware:** Our approach should not rely on any specialised hardware that is not found in off-the-shelf systems, allowing service providers to use commodity hardware for their servers.

D-2 **Commodity user hardware:** Users should not have to upgrade their device in order to use a prediction service that uses an ideal implementation of our approach.

3.5 Assumptions

For our approach we make the following assumption that cannot be taken for granted:

1. The source code, that computes a prediction given an input and a model, as well as its configuration is publicly available through trusted thrid party. This allows users to create a reproducible build. Thus, users can measure the code themselves, enabling them to compare their own measurement with the measurement received from the service provider.

2. We assume that a model used by the service provider is adequate for its purpose. Therefore, we do not cover the scenario in which a malicious model provider tricks his users by using an inferior model, or no working model at all.

Having made these assumptions, we argue, however, that we can do so without loss of generality. Current state-of-the-art frameworks, such as PyTorch[1] and Caffe2[2], are developed as open source projects where everybody can study the source code or even contribute to future releases. These projects also include detailed instructions to build their binaries from source as well as offer pre-built binaries. Moreover, due to their open source nature, independent parties are able to audit the source code of these frameworks and present their findings to the public as well as the developers. In addition, we consider the threat of service providers using inferior models as neglectable, as users can usually choose between multiple competing service providers. Therefore, providing bad or inaccurate predictions would be a bad business decision for service providers.

[1]https://pytorch.org/
[2]https://caffe2.ai/

4. Design

In this chapter we first give a general overview of our proposed solution. Afterwards, we focus on the different components of our design and cover them in detail.

4.1 Design Overview

Figure 4.1 depicts a broad overview of the system, showing how different elements of our design interact with each other. The *client* communicates directly with

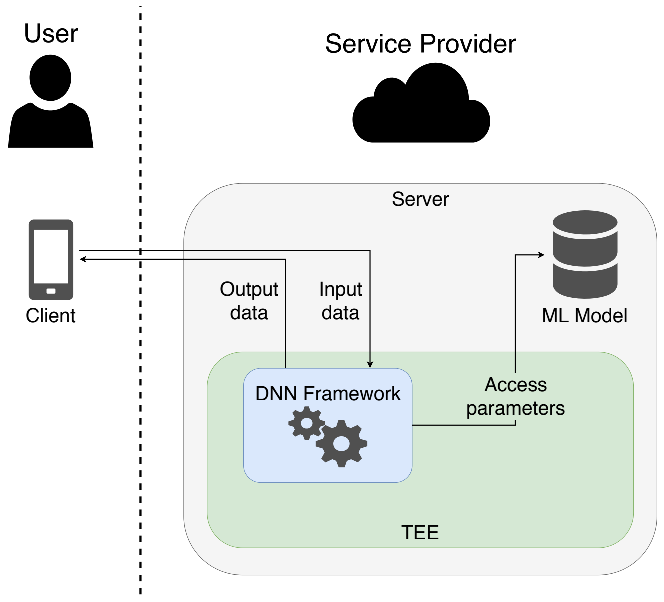

Figure 4.1. Overview of the complete system

the TEE running on the *server* of the prediction service. The REE, or *untrusted environment*, describes the environment outside of the TEE, or *trusted environment*. The REE runs the *untrusted application* which only acts as a gateway to forward encrypted queries and their respective encrypted predictions. Once the *trusted application* running inside the TEE receives a new query, it computes the appropriate prediction using a DNN framework. Although all computation is done

inside the TEE, the used machine learning model is stored outside the trusted environment to keep the trusted computing base as small as possible.
Even though figure 4.1 depicts the client as a mobile phone, it is representative for any kind of end-user device with enough computational power. In order to save bandwidth, the client pre-processes the query before sending it off. In the example of an image, this means reducing the pixel dimensions and possibly the number of colour channels. In addition, this pre-processing step helps reducing the workload of the trusted application, as inputs are already downscaled by client devices. In order to not leak any information regarding the model, this pre-processing step must be independent from the deployed model.

4.2 Prediction Service

The server-side prediction service consists of two main components, trusted and untrusted application.
The untrusted application launches first when starting the server. Its purpose is to launch as well as initialise the trusted application and to perform necessary pre-processing steps. First, it parses the model into a form the trusted application can interpret. Afterwards, the untrusted application extracts the model parameters to save them in the untrusted environment and make them available for the trusted application. The reason for this is the potentially large amount of memory necessary to store all parameters. For instance, VGG-19, a model for large scale image recognition designed by Simonyan *et al.* [50], consists of 144 million weights, resulting in a total model size of 575 MB. As TEEs are designed to reduce the trusted computing base to a minimum, we only want to store necessary data in the trusted environment. Furthermore, we consider the model as public in the context of the server, meaning there is no need to protect its contents. Besides launching and initialising the trusted application, the untrusted application is also responsible to act as a gateway, allowing users to interact with the trusted application.
The trusted application, which forms the second component of the prediction service, is in charge of computing predictions for given queries. Given the user's input, the trusted application decrypts the query using a shared key and possibly pre-processes the input data to match the requirements of the model. Afterwards, it traverses the deployed model layer by layer. The trusted application also performs necessary checks to avoid buffer overflows. For instance, it verifies whether two matrices have compatible dimensions for a given operation. In case of an error, the trusted application stops execution and reports the encountered error to the

untrusted application without specifying its details. Whilst the model parameters are stored in untrusted environment, all intermediary and final results are stored inside the TEE to ensure no user data leaks to the service provider. Keep in mind that a neural network is a sequence of matrix operations, meaning outputs of all layers are stored in the form of matrices, as described in section 2.1. Ultimately, the final model output is encrypted, using the same symmetric key that was used for decrypting the query, and sent back to the untrusted application which forwards it to the client.

4.3 Client Application

The design of our client application is fairly simple. First, the client application produces input data that will later be passed to the prediction service in form of a query. The kind of data and how it is produced depends on the use case and the type of prediction service. For instance, in the use case of an image recognition service, the user could take a picture with a smart device. Alternatively, the user might select her favourite films, generating a text based input of film characteristics. Once the input is generated, it is encrypted using a shared symmetric key, which is only known by the client and the trusted application. Afterwards, the encrypted input is wrapped by a query that might contain additional data. The amount and type of data that a query adds to the encrypted input data depends of the use case and the implementation itself. For instance, the query might, in addition to the actual payload, contain parameters needed for authentication so a user can be billed appropriately.
After the prediction service has responded with the encrypted output data, the same symmetric key that was used for encrypting the input data is used to decrypt the prediction.

4.4 Client-Server Communication

Defining the communication between client application and prediction service is crucial for our design in order to provide secrecy, integrity, and performance. A complete overview of the communication design is depicted by figure 4.2. In order to create an authenticated channel and avoid communication overhead, we use the *0-round-trip attestation* designed by Krawiecka *et al.* [31]:
Whenever the trusted application is launched, the TEE first measures its contents, i.e. the code that it is executing. The source code as well as the configuration producing this code is publicly available through a trusted third party, as explained

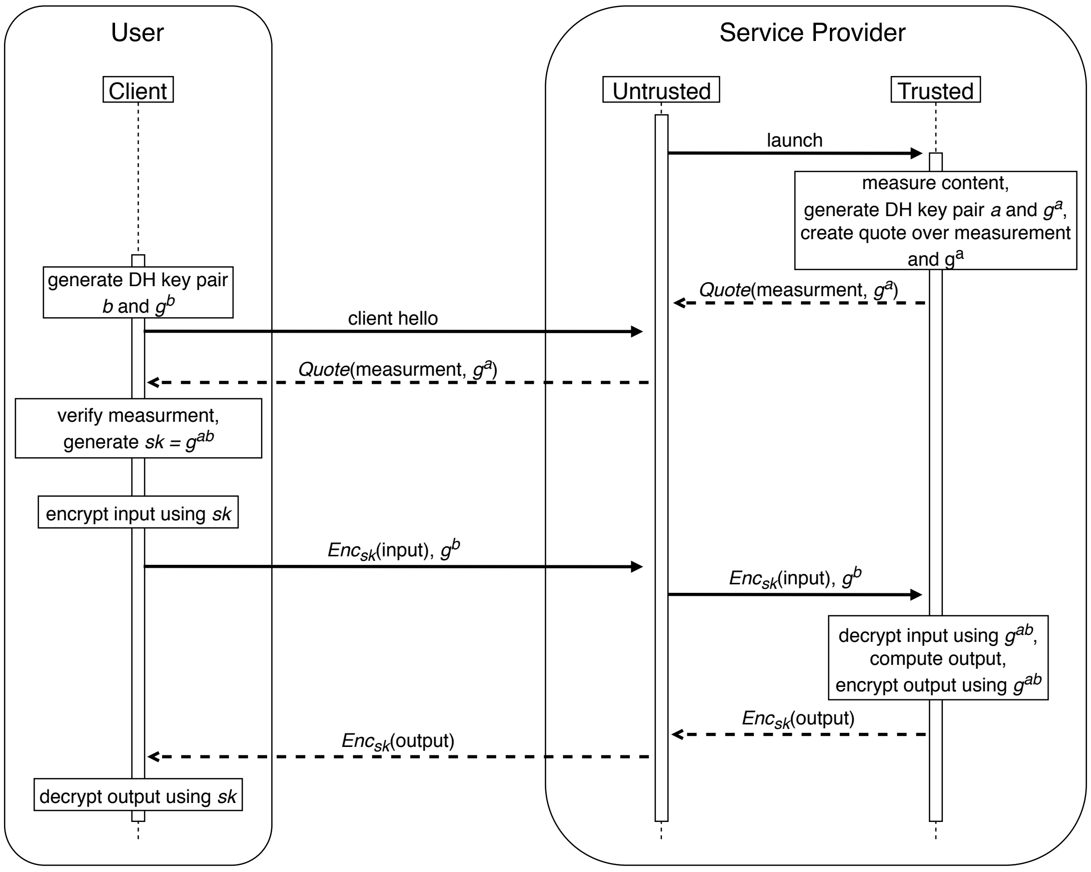
User
Service Provider
Client
Untrusted
Trusted
launch
measure content, generate DH key pair a and g^a, create quote over measurement and g^a
Quote(measurment, g^a)
generate DH key pair b and g^b
client hello
Quote(measurment, g^a)
verify measurment, generate $sk = g^{ab}$
encrypt input using sk
Enc_{sk}(input), g^b
Enc_{sk}(input), g^b
decrypt input using g^{ab}, compute output, encrypt output using g^{ab}
Enc_{sk}(output)
Enc_{sk}(output)
decrypt output using sk

Figure 4.2. Communication between client and server

in section 3.5. After it has been measured, the trusted application generates a new Diffie-Hellman (DH) key pair consisting of public key g^a and private key a. The initialisation process is completed by the TEE generating a quote over the measurement of the trusted application as well as g^a and returning the quote to the untrusted application. This quote represents an unforgeable representation of the code running inside the trusted application, enabling the trusted application to remotely attest its legitimacy. In addition, the quote is bound to a public key g^a. As a result, any input data that was encrypted using g^a, or any key that was derived from g^a, can only be decrypted by the trusted application for which the quote was generated. Any changes to the code, for instance to export a to the untrusted application, will be reflected in the measurement.

The client, on the other hand, generates her own DH key pair, public key g^b and private key b. At this point the client can request the quote of the trusted application. The client can then verify the validity of the measurement and decide whether it wants to trust it. Moreover, g^a allows us to derive a shared key $sk = g^{ab}$ that is used to communicate with the trusted application over an authenticated channel.

The steps explained so far conclude the bootstrapping process, meaning it only has to be done once and possibly well ahead of any user interactions. From this point onwards, whenever the client sends new input data to the service provider, it encrypts the data using the shared key sk. In addition to the encrypted input data, the client also sends its public key g^b, enabling the trusted application to derive sk and decrypt the sent input data. The public key g^b is sent with every query, allowing the trusted application to remain stateless and clients to generate a new key pair whenever they see fit. Alternatively, the trusted application could cache sk to improve performance. Similar to input data, output data is encrypted by the trusted application using sk and sent to the client where it is decrypted using the same key.

5. Implementation

In this chapter we discuss our proof-of-concept implementation of the server-side prediction service. The client-server communication, as described in section 4.4, is modelled after the communication scheme proposed by Krawiecka *et al.* which they implemented for SafeKeeper [31]. We decided to use Intel Software Guard Extensions (SGX) for our implementation due to its ease of use and the wide availability of supported chip sets [18]. In addition, SGX provides remote attestation capabilities which is necessary for our communication design. Nonetheless, any TEE that supports remote attestation could be used to implement the design proposed in section 4.2. Our proof-of-concept implementation conforms to the `C++11` standard, as this is the latest standard supported by SGX [25]. Furthermore, we decided to only accept machine learning models in ONNX format. This gives our implementation more flexibility since we do not restrict model designers in what tools or frameworks to use.

5.1 Untrusted Application

As described in section 4.2, the untrusted application functions as a gateway for encrypted input as well as output data. In addition, the untrusted application also initialises the enclave, i.e. the trusted application, and performs necessary preprocessing steps. We use the 0-round-trip library[1] of SafeKeeper [31] to request the quote from our enclave. This quote is used for key derivation and remote attestation as discussed in section 4.4. Once the untrusted application receives the quote, it is then stored by the untrusted application from where it can be sent to clients.

Model Parsing

If the quote was successfully created, the untrusted application will load and parse the ONNX model file it was instructed to use. This file is stored in plaintext in

[1]https://github.com/SSGAalto/sgx-utils

untrusted memory. The ONNX format uses *Protocol Buffers*[2] (protobuf) as its underlying file structure. Due to dependency issues of the protobuf library, we cannot use it inside our enclave. Therefore, we developed our own SGX-compatible ONNX model representation which can be used by our enclave. We will cover these issues as well as our own ONNX library in section 5.3 in greater detail. In general, we use the protobuf library to convert the ONNX model into our own model representation using `C++` objects. Before passing the ONNX model to our enclave we have to extract its parameters into a separate `float` buffer that stays in untrusted memory, as explained in section 4.2. We limit our implementation to only supporting `float` parameters for simplicity reasons, the overall design of our proof-of-concept implementation can support arbitrary data types. Whilst extracting model parameters, we create a `std::unordered_map` to keep track of parameter offsets. We use parameter names as keys to store the offset in the `float` buffer of the corresponding parameter. We chose an unordered map over other data structures, such as an ordinary `std::map` or a `std::vector`, as it internally uses a hash map to store elements. Thus, `std::unordered_map` provides the best average element lookup performance of $\mathcal{O}(1)$.

Passing Data to the Enclave

After all parameters have been extracted, we can pass our model to the enclave. However, SGX only provides a `C` interface for passing data to enclaves [25]. Thus, we cannot directly pass our `C++` objects. A possible solution for this would be to rely on `C` structs. But as flexible arrays are not supported by Enclave Definition Language (EDL) syntax, passed structs have to be of fixed size [25]. Hence, we would either have to define very large structs that can fit even extensive amounts of data or define the same structs in multiple sizes to support flexible model sizes. Whilst using large, fixed-size structs potentially wastes resources for small models, defining structs multiple times in different sizes adds unnecessary complexity. In order to avoid these drawbacks, we chose to use serialisation to pass model data to the enclave. We serialise a given model to a single, large `uint8_t` buffer, which allows us to only use exactly the amount of data we need. The details of how we serialise our `C++` objects are described in section 5.3.3. Besides the serialised model, we also pass a pointer to the parameter buffer as well as the corresponding serialised offset map to the enclave. When passing our parameters to the enclave, we have to instruct SGX which parameters should be copied over into trusted memory and which should remain in untrusted memory. We can achieve this by leveraging EDL syntax. We label the pointer to the buffer containing all exported model parameters as `user_check` parameter to indicate it should remain

[2]https://developers.google.com/protocol-buffers/

in untrusted memory. All other parameters, i.e. pointers to buffers containing the serialised model and the serialised offset map, are labeled as `in` parameters, meaning they will be copied into trusted enclave memory [25].

Receiving Output Data

After the enclave initialisation process is completed, the untrusted application simply sends encrypted input data as `uint8_t` buffer labeled as `in` parameter to the enclave via ECALL. In addition, the untrusted application also has to provide a buffer for encrypted output data, labeled as `out` parameter. This tells SGX that this buffer will be filled by the enclave. However, the buffer size must be known in advance, when making the ECALL. We can easily calculate the necessary buffer size as each model in ONNX format must define the shape of its output. Moreover, for a given model, we only have to do this calculation once as the output shape and therefore the buffer size will not change for different input data.

5.2 Enclave

Although the enclave forms the core of our approach, its implementation is rather simple. Once the enclave has been started and initialised, as described above, it is ready to receive the model it is meant to use for its calculations.
All necessary model data is passed to the enclave via a single ECALL. This reduces further interactions between untrusted application and enclave, i.e. ECALLs and OCALLs, to exchanging encrypted input and output data. Consequently, after the enclave is passed input data, the CPU does not have to switch between enclave and non-enclave mode. The reason why we want to avoid switching CPU mode is due to the performance overhead this introduces. Enclave exit and entry instructions both cause the CPU to perform multiple operations, including flushing certain cache entries, saving return pointers, and updating internal control structures [36].
After de-serialising the passed model data, the resulting model is stored by the enclave, enabling it to receive input data. Note that the model parameters, which require most of the memory of a given model, remain in untrusted memory. The model that is stored in enclave memory only consists of the model graph with all its nodes and attributes. Parameter values necessary for execution are referenced through pointer variables. We cover our model representation in greater detail in section 5.3.1. For every given blob of encrypted input data, we decrypt the serialised input data, using the shared key as described in section 4.4, de-serialise the input data, and run the model with said input data. After the computation finishes, we first check whether the computation was successful. In case any errors occurred, we report the error back to the untrusted application without

providing any output data. Otherwise, we serialise the output data, encrypt it, copy it into the provided output buffer, and indicate to the untrusted application that all operations where successful.

5.3 DNN Framework

We briefly mentioned in section 5.1 that we implemented our own ONNX library due to dependency issues with protobuf and SGX. This issue also arises when attempting to use existing machine learning frameworks such as Caffe2, PyTorch, and TensorFlow. These frameworks have numerous dependencies, many of which cannot be used by enclaves, such as system calls or dynamic libraries in general [25]. Thus, it would require extensive work to port any existing framework in order to make it compatible with SGX. Moreover, existing frameworks were not designed for using as little memory as possible but for performance. In fact, Caffe2, PyTorch, and TensorFlow all offer pre-compiled Anaconda[3] packages. When downloaded and decompressed, these Anaconda packages for Linux[4] have the following sizes:

- Caffe2: 72.8 MB
- PyTorch: 187.4 MB
- TensorFlow: 261.5 MB

If we recall that with current (October 2018) SGX hardware the Enclave Page Cache (EPC) of SGX can only allocate about 90 MB for all enclaves before paging is introduced [27], it becomes obvious that existing machine learning libraries should not be used by an enclave.
As a result, we created an SGX compatible machine learning framework for deep neural network predictions from scratch. The framework consists of two libraries, a trusted and an untrusted library. The untrusted library is meant for the untrusted application, allowing it to parse given ONNX files and translate it to our model representation. In addition, the untrusted library has logging functionalities for models. The trusted library only consists of the core code that handles model representation and implements different machine learning operators. After compilation, our libraries have the following sizes:

- trusted library: 2.4 MB

[3]https://anaconda.org/
[4]Package sizes for other operating sizes might differ

- untrusted library: 2.6 MB

Note, however, that at the moment of writing this thesis (October 2018) we have only implemented a small subset of the total operator set that existing frameworks offer. We cover which operators we implemented in section 5.3.4. Moreover, we only support forward passes, meaning our framework is meant for model execution but not model training. Nonetheless, we expect our framework to be significantly smaller in terms of raw byte size than other existing frameworks. During the process of writing this thesis, Eigen[5], a popular C++ library for linear algebra, was ported by Tramèr *et al.* [51] to be used inside an SGX enclave. The Eigen library is extensively used by multiple state-of-the-art machine learning frameworks, such as Caffe2 and TensorFlow. For our library, we do not use the SGX compatible port of Eigen, as it was released after we finished our proof-of-concept implementation.

5.3.1 Model Components

We chose to design our model representation after ONNX and therefore we also represent a model as an acyclic graph where each node represents a single, atomic operation. In addition, we used many design principles found in Caffe2 and Darknet [45]. For the rest of this section, we will cover the different components of our framework. We start from the most low-level components and move to more general components.

Tensor

The `Tensor` component stores the data that other components use for their computations. Input as well as output data, model parameters, and intermediary results are all represented as `Tensor` objects. Similar to the Tensor component of ONNX, a `Tensor` object has three important properties: name, values, and shape. The name of each `Tensor` object must be unique, as defined by the ONNX specification [42]. This allows us to easily find existing `Tensor` objects during runtime. Values are stored in a typed, one-dimensional `std::vector` object, meaning each `Tensor` object is also typed. Finally, the shape is stored in a one-dimensional `std::vector<int>` object. We purposely decouple the values and their shape, i.e. their dimensionality, to increase simplicity and flexibility.
In addition to maintaining its own values, the `Tensor` component also supports wrapping around existing values, making said values available through the `Tensor` object. We implement this functionality by using the `std::reference_wrapper` template provided by `C++`. This is used for model parameters, as they are stored

[5]http://eigen.tuxfamily.org/

in untrusted memory, whilst all computation takes place in trusted memory. Whether or not a `Tensor` maintains its own values is hidden behind the function `get_mutable_data()`, which returns a pointer to an object's value array. Which pointer is returned is handled internally by the `Tensor` object and not visible to the calling code. Moreover, we designed all machine learning operations to be pure, i.e. free of side effects. As a result, existing `Tensor` objects are treated as read-only, making sure we do not leak data by writing to untrusted memory.

TensorInfo

The `TensorInfo` component is the blueprint of a `Tensor`, describing its name, data type and shape. Generally, the `TensorInfo` component is based on the ValueInfo component of ONNX described in section 2.1.4. Therefore, `TensorInfo` objects, just like ValueInfos, are used in places where we need a placeholder for a `Tensor` that is not available yet.

TensorContainer

The `Tensor` component is implemented as a templated class, meaning a `Tensor` object is always typed, e.g. `Tensor<float>` or `Tensor<int>`. In order to store tensor data without strict typing, we define a `TensorContainer` component. A `TensorContainer` object stores exactly the same data as a `Tensor` object would, i.e. its name, shape, type and values. However, instead of storing values in a typed `std::vector` object, the `TensorContainer` component stores the raw bytes of all values in a single `std::string` object. Whenever we need access to the `Tensor` that a `TensorContainer` represents, we export the `Tensor` object with a single function call. The `TensorContainer` component allows us to store `Tensor` objects of different types together in the same container object, such as a `std::vector`. In addition, by using the `TensorContainer` component, we can define container objects that can store typed tensor data without having to know the data type in advance. For example, we can define a `std::vector<TensorContainer>` variable without having to know what data types the underlying `Tensor` objects might have.

NodeAttribute

As the name suggests, the `NodeAttribute` component defines an attribute of a given node, i.e. a given operation. This component is closely modelled after the Attribute component of ONNX described in section 2.1.4. In order to be able to support storing different data types, the `NodeAttribute` component defines multiple member variables ranging from a single `float` or `int` over `std::vector<float>` and `std::vector<int64_t>` to `std::string`. As opposed to Attribute component defined by ONNX, our `NodeAttribute` component defines an `AttributeType` member variable. This variable describes the purpose of a given `NodeAttribute` object. We chose

`AttributeType` over `std::string` to save memory.

Node

The `Node` component is based on the Node component found in ONNX, meaning it stores the same data we described in section 2.1.4. We store attributes in a `std::map<AttributeType, NodeAttribute>` object which allows us to quickly find a certain `NodeAttribute` object when it is needed. The type of a `Node` is stored in a `OperatorType` variable. The most important function the `Node` component defines is the `forward` function. It accepts a list of input `Tensor` objects as well as a list where the `Node` can store its output `Tensor` objects. The `forward` function is implemented as a pure virtual function, making the `Node` component an abstract class. Therefore, each operation, such as Convolution, ReLU, or MaxPool, is implemented as a separate, derived class that inherits from `Node`. These derived classes each implement their own `forward` function that performs the actual operation.
The `Node` component stores its inputs as a `std::vector<std::string>`. Moreover, if we recall that all `Tensor` names must be unique, it becomes clear why storing the name of a `Tensor` is sufficient to locate it at runtime. As a result, the calling code can retrieve all inputs a `Node` defines and provide it with the corresponding `Tensor` objects. Although the computation differs greatly between the different implementations of operations, they all share a common execution flow:

1. Verify that the amount of passed input `Tensor` objects matches the expected amount.
2. Ensure that the passed inputs are compatible with each other, e.g. have compatible shapes, and that all necessary `NodeAttribute` objects are present. Due to how SGX handles errors, we cannot simply fail when constructing a derived `Node` object by throwing an exception. As a result, we perform these check during execution and abort if needed.
3. Acquire pointers of the underlying raw data of input `Tensor` objects using `get_mutable_data()` and perform operation-specific computations using said pointers.
4. Store calculated output `Tensor` objects in the provided list, so that that they become available to the calling code.

Graph

The top-level component of our framework is the `Graph` component. This component is closely modelled after the Graph component of ONNX described in section 2.1.4, meaning it stores the same data but represented by our own components. The initialisers, i.e. a list of constant `Tensor` objects, are stored in an `std::unordered_map` that uses `std::string` objects, i.e. the names of the stored `Tensor` objects, as keys. Initially, when a `Graph` object is constructed, the `std::unordered_map` only accommodates trained model parameters. As we traverse we `Graph` object, however, this `std::unordered_map` object is filled up with intermediary results which serve as inputs of future `Node` objects.

5.3.2 Model Execution

Due to the memory limitations in current SGX hardware [27], we want to be as memory efficient as possible. As `Tensor` objects consume the most memory, we want to ensure each `Tensor` objects is only created once and only accessed through pointers or references afterwards. We achieve this by wrapping all `Tensor` objects in `std::shared_ptr` objects, after which we access `Tensor` objects only through their respective shared pointer. A `C++` shared pointer keeps a reference counter. This counter is increased whenever a shared pointer is passed as a parameter and decreased when the pointer runs out of scope. Once the reference counter of a shared pointer reaches zero, the pointer deletes itself, including the data it points to. This design of shared pointers makes memory management during model execution straightforward.
During execution, we store all shared pointers that contain `Tensor` objects in a `std::unordered_map` object that uses `std::string` objects as keys, as explained in section 5.3.1. Moreover, we mentioned in section 5.3.1 that the name of a `Tensor` object must be unique. Thus we use the name of each `Tensor` object as a unique key, allowing us to efficiently locate `Tensor` objects at runtime. We populate this `std::unordered_map` by inserting all outputs returned by the `forward` function of each `Node` object in our `Graph`.
The ONNX specification states that the list of nodes of a graph must be topologically ordered [42]:

Given a graph G with a topologically ordered list of nodes $L = \{X_1, X_2, ..., X_n\}$ and two nodes $X_i, X_j \in L$ with $i < j \leq n$. Then no inputs defined by X_i must refer to outputs defined by X_j.

We exploit this property and retain the topological order whenever we translate a given ONNX model to our model representation. As a result, we can safely assume that during model execution every input that a current node defines must exist in our `std::unordered_map` object. If a requested input does not exist, we have encountered an unresolvable problem and abort execution. After we have successfully traversed the list of `Node` objects of a `Graph`, we retrieve the output `Tensor` object the `Graph` defines from the map and return to the calling code.

5.3.3 Serialisation

We heavily rely on serialisation when passing data between untrusted application and enclave. Consequently, each component described in section 5.3.1 defines its own serialisation function as well as a constructor for de-serialisation. The intuition behind our serialisation scheme is straightforward: store the size of a serialised data blob, followed by the data blob itself. In addition, data blobs might follow a hierarchy. Our scheme is illustrated by figure 5.1. For example, a `Graph`

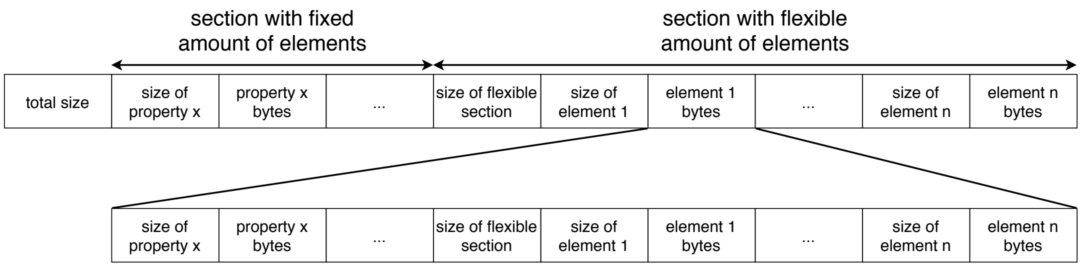

Figure 5.1. Serialisation scheme for passing data to the enclave

component contains a list of `Node` components. Each `Node` defines a list of inputs and `NodeAttribute` components it requires in order to compute its defined outputs. As a result, when serialising a `Graph` we have to recursively traverse its hierarchy until we reach trivial types, such as `int` or `float`, or arrays and vectors consisting of trivial types. Once we have reached this point, we convert the data to `uint8_t` buffers. After all trivial types have been converted, we backtrack one level at a time and always merge `uint8_t` buffers from previous levels as we move up. We continue this until all data has been converted and we are left with one large `uint8_t` buffer.

We follow a similar approach when we de-serialise data. Continuing with the `Graph` component example, we can easily determine where specific data sections, like the list of `Node` components and even individual `Node` components within that list, start as well as end. Therefore, when arriving at a new data section, we simply read its length and pass the specified data range to the appropriate component constructor. We know which constructor to call because we de-serialise objects in the same order as we serialise them.

5.3.4 Implemented Operators

For our proof-of-concept implementation, we implemented the following operators as defined by *version 6* of the default ONNX operator set [43]:

Add

The *Add* operator performs an element-wise addition of two input tensors A and B.

Add supports two attributes $axis$ and $broadcast$, which allow B to be broadcasted to match the shape of A. Setting $broadcast = 1$, indicates B should be broadcasted and $axis$ can be used to decide along which axis. If $axis$ is not set, suffix matching is assumed, meaning the shapes of A and B are compared starting with their last value. For example:

- Consider A has shape $2 \times 3 \times 4 \times 5$ and B has shape 3×4. Then A and B can be added with attributes $broadcast = 1$ and $axis = 1$, meaning B is broadcasted to shape $2 \times 3 \times 4 \times 5$.
- Consider A has shape $3 \times 4 \times 5 \times 6$ and B has shape 5×6. Then A and B can be added with attributes $broadcast = 1$, meaning B is broadcasted to shape $3 \times 4 \times 5 \times 6$.

Constant

The *Constant* operator simply outputs the tensor it was supplied via its $value$ attribute. Whenever we encounter a Constant operator while parsing an ONNX model, we drop the operator itself and add the tensor specified by its $value$ attribute to the list of initialisers of our `Graph` component we mentioned in section 5.3.1.

Conv

The *Conv* operator computes the same operation as the Convolutional Layer described in section 2.1.3. Thus, this operator takes an input tensor X, a weight tensor W filled with filters, and an optional bias tensor B as inputs. In addition, Conv defines the attributes $dilations$, $kernel\ shape$, $pads$, and $strides$. The $kernel\ shape$ attribute defines the height and width of the filters, or kernels, stored in W. If omitted, the value of $kernel\ shape$ is derived from W. Both $dilations$ and $strides$ define how each filter is moved over the input X. If not specified, both attributes default to 1. Lastly, the $padding$ attribute defines with how many rows of zeros X should be padded along its height and width axes. This is needed if the filters would otherwise "overflow" at the edges of X. The $padding$ attribute is optional and its default value is 0, meaning no padding is added.

Gemm

The *Gemm* operator computes a General Matrix Multiplication (GEMM). Thus the Gemm operator takes three input tensors A, B, as well as C and computes output Y as follows:

$$Y = \alpha * A * B + \beta * C \tag{5.1}$$

Furthermore, Gemm defines the attributes α, β, $broadcast$, $transA$, and $transB$. Both $alpha$ and $beta$ describe the scalar values of the equation above and default to 1. The attributes $transA$ and $transB$ indicate whether A and B should be transposed before calculating output Y. The $broadcast$ attribute defines whether C should be broadcasted. Considering input A is of shape $M \times K$ and input B has shape $K \times N$, both input C and output Y must have shape $M \times N$. Thus, $broadcast$ can be used to broadcast C to shape $M \times N$. The Gemm operator represents a fully connected layer, as described in section 2.1.3, with the sole difference that no activation function is applied. To model a fully connected layer, we have to append the operator of the desired activation function to the Gemm operator.

MatMul

The *MatMul* operator calculates the matrix product of two input tensors A and B. Depending on the shapes of A and B, the operator behaves as follows:

- If A and B are 2-dimensional, the operator computes an ordinary matrix multiplication.
- If either input A or B is n-dimensional with $n > 2$, said input is treated as a stack of 2-dimensional matrices. The other input is broadcasted accordingly to match the shape of the n-dimensional input.
- If A is 1-dimensional, said input is promoted to matrix by prepending a 1 to its shape. After the matrix multiplication is completed, the prepended 1 is removed from the shape of A.
- If B is 1-dimensional, said input is promoted to matrix by appending a 1 to its shape. After the matrix multiplication is completed, the appended 1 is removed from the shape of B.

MaxPool

The *MaxPool* operator computes the same operation as the MaxPool Layer described in section 2.1.3. It accepts one input tensor X and defines the attributes $kernel\ shape$, $pads$, and $strides$. All attributes have the same purpose as with the Conv operator. Therefore, $kernel\ shape$ defines the size of the filter that is moved across X, $strides$ defines how the filter is moved, and $pads$ denotes with how many rows of zeros X should be padded along its height and width axes.

ReLU

The *ReLU* operator element-wise computes the ReLU activation function on its input tensor as defined in section 2.1.3.

Reshape

The *Reshape* operator reshapes its input tensor to the shape specified by its $shape$ attribute. The $shape$ attribute must not change the overall size of the input. Each value of $shape$ that is equal to zero, remains unchanged. Furthermore, at most one value of $shape$ may be -1, meaning the actual value is inferred by the remaining values of $shape$. For example, given an input tensor with shape $2 \times 3 \times 4$:

- a shape of $0 \times 2 \times 2 \times 3$ would output a tensor of shape $2 \times 2 \times 2 \times 3$
- a shape of 3×-1 would output a tensor of shape 3×8

Sigmoid

The *Sigmoid* operator element-wise computes the Sigmoid activation function on its input tensor as defined in section 2.1.3.

TanH

The *TanH* operator element-wise computes the TanH activation function on its input tensor as defined in section 2.1.3.

5.3.5 Avoiding Memory Side-Channel Attacks

For all operator implementations of our proof-of-concept implementation, we designed our code such that memory access patterns for both writing and reading operations remain *input-independent*. This is necessary, as SGX is prone to side-channel attacks. For instance, attacks have been found that reveal enclave secrets by monitoring cache accesses of read and write operations made by enclave code [39][5][53]. Fortunately, most operations discussed in this thesis have an input-independent control-flow which results in input-independent cache access patterns. However, in the case of the ReLU and MaxPool operators, control-flow depends on the input that we are trying to protect.

In the case of ReLU, we have to traverse the complete input tensor and determine for each individual value whether it is larger than zero. Thus, read operations are already input-independent as we always have to read the complete input tensor. New tensor objects are initialised with zeros as default values, thus we could only write a value to the output tensor if it is larger than zero. However, this would leak information about the input, as an attacker monitoring our write operations could see when we write a value to our output tensor and when not. To mitigate this we always write a value to our output tensor, even if that value is zero. This way, we hide if an input value is smaller or larger than zero as write operations are identical for all inputs.

The MaxPool operator slides a filter of pre-defined size across its input tensor and copies the maximum value of each filter to its output tensor. Similar to ReLU, read operations are already input-independent, as we always have to read the entire input tensor regardless of its values. For our implementation, we use a straightforward algorithm for finding the maximum value of every filter: we use a helper variable $local\_max$ that stores the current maximum which, once we have seen every value within a filter, holds the overall local maximum value that is copied to the output tensor. Thus, if we would only update $local\_max$ if we find a larger value, an attacker monitoring our write operations could infer which values in our input tensor are larger than others. Therefore, we update $local\_max$ for every value with either a new, larger value or the current value of $local\_max$. Thus, write operations are identical for all inputs, regardless of their values.

6. Evaluation

We base our evaluation on the requirements and goals defined in chapter 3. Consequently, we evaluate security, functionality, performance, and deployability of our proof-of-concept implementation individually. For our empirical tests, we consider the following settings:

Plaintext, untrusted memory: Input and output data remains unencrypted at all times and our proof-of-concept implementation is executed by an untrusted application.

Plaintext, SGX enclave: Input and output data is unencrypted and our proof-of-concept implementation is executed inside an SGX enclave.

Encrypted, SGX enclave: Input data is provided in encrypted form and output data is also encrypted. Consequently, input data is decrypted before computation begins and output data is encrypted before it is returned. In addition, our proof-of-concept implementation is executed inside an SGX enclave.

6.1 Security

The security requirements we defined in section 3.3.1 have the highest priority, together they form the main drive behind this thesis. For our evaluation we use a combination of empirical and theoretical analysis. In addition, we consider existing attacks and how they might affect our system.

Model Confidentiality (S-1)

In our proposed design and its proof-of-concept implementation, the machine learning model of the service provider is stored on the server. Thus, users, malicious or not, never have direct access to the model and its data. However, users have oracle access to the model, meaning for any given input data the model will produce output data that is sent back to the appropriate user. A malicious user can exploit this, as she controls what data is sent to the model. Furthermore, she can time the response time.

Assuming a malicious user is able to predict the latency of the network over which she is communicating with the enclave, she can infer the overall complexity of the used model. Figure 6.1 depicts the average times needed for executing the graphs of two different models inside an SGX enclave. The two models used here serve no other purpose than to highlight how model complexity affects the overall computation time. Whilst *Model A* only consists of 3 nodes that all compute operations on a small input of dimension $1 \times 1 \times 28 \times 28$, *Model B* comprises 10 nodes with significantly more parameters and expects an input with dimensions $1 \times 3 \times 224 \times 224$. The executed model graphs are presented in appendix A. We can

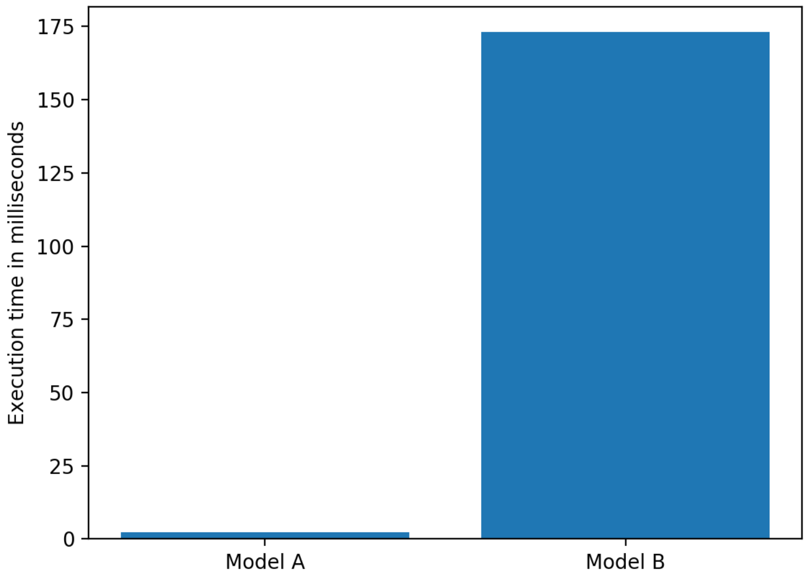

Figure 6.1. Execution times of different models inside an SGX enclave

easily see that more complex graphs need more time to compute a prediction for a given input. Note, however, that "complex" in this setting does not necessarily refer to the structure of the graph, i.e. how many nodes it is comprised of, but can also refer to single operations. For instance, a graph that only consists of a single node which computes the product of two large matrices will take longer to execute than a graph comprised of tens of nodes that all compute simple operations, such as additions, or operate on smaller inputs. Apart from a malicious user, the model execution time can also be observed by a malicious outsider, who can timestamp packets sent between user and service provider. To mitigate this attack, we could add a, potentially randomised, delay for providing output data. This way a malicious user or malicious outsider cannot infer the used model from measuring the response time.

Besides timing the response time, a malicious user might also mount a model extraction attack. Tramèr *et al.* [52] developed an attack where they exploit the received output of a deployed model to build a local copy of said model. For their attack, they only rely on the publicly available interfaces granted by the service provider. Tramèr *et al.* apply their technique to different models trained on four different data sets. They report to be able to extract a deployed model using between 300 to 1500 queries. Juuti *et al.* [28] further improved on this attack, achieving a much higher agreement rate of up to 88.8% between deployed model and local copy. Our design and its proof-of-concept implementation do not address this attack and are therefore prone to model extraction attacks by a malicious user. However, Juuti *et al.* and Tramèr *et al.* both describe countermeasures to mitigate their attacks. As a malicious outsider cannot see the input and output data sent between user and service provider, he cannot mount a model extraction attack on its own or benefit from an attack carried out by a malicious user.
We conclude that our design combined with the defences mentioned above is able to protect a deployed model. In fact, Hanzlik *et al.* [19] implemented precisely these defences, among others, to further protect a model used by an SGX enclave for on-device machine learning. As a result, we note that executing the model inside a TEE does not negatively affect model confidentiality.

User Privacy (S-2)

Due to the design of our system, all input and output data is encrypted using a shared key only known to the user and the enclave. As a result, the untrusted application, which the service provider controls and whose code is not visible to the user, can only see the unencrypted meta data of a user query. The plaintext data that a query contains, as described in section 4.3, is application specific. It might contain information about the user needed for authentication, as well as general data such as timestamps. The untrusted application can, however, monitor the enclave as it is running and even mount a side-channel attack.
Applications running inside an SGX enclave are potentially prone to different side-channel attacks:
Cache Attacks. Moghimi *et al.* [39] as well as Brasser *et al.* [5] have shown *Prime+Probe* attacks that can successfully recover AES and RSA keys, respectively, that were used by an SGX enclave. For their attacks, they monitored hundreds of decryptions in order to reveal the used key. Brasser *et al.* also show that an SGX enclave can leak non-cryptographic data. They were able to reveal the provided input of a genomic processing enclave using their attack. The general intuition of a Prime+Probe attack is to first prime the cache, wait for the victim application to execute and then probe the cache to identify which cache lines were

operator	input-independent reads	input-independent writes
ReLU	yes	no
MaxPool	yes	no

Table 6.1. Operators vulnerable to cache attacks

used by the victim application. As a result, the attacker, knowing the source code of the victim application, can infer what data was processed based on the control flow and observed cache accesses. The control flow of the enclave of our proof-of-concept application is mostly based on the provided model, i.e. its nodes and their attributes. The exceptions to this are shown in table 6.1. Both operators rely on either comparing input values with one another or a fixed value. Thus, read operations result in identical cache accesses for all inputs as all input values must always be read. However, both operators could leak information about input values due to their write operations. We mitigate this issue by adding "dummy" writes, i.e. for each input value we perform a write operation even if it is unnecessary. We describe the mitigations for these operators in greater detail in section 5.3.5. Consequently, the memory access pattern will be identical for all input data for every operation. Therefore, a malicious service provider cannot use cache attacks to reveal input or output data. In addition, we use *AES-CTR* encryption and decryption operations provided by the SGX cryptography library. These functions use special hardware instructions and other software based techniques to mitigate cache attacks [4].

Page Table-Based Attacks. SGX enclaves can leak sensitive information based on their page accesses. Xu *et al.* [53] have shown that they can reveal text as well as image data by tracking data page accesses of trusted applications. As we explained above, our proof-of-concept implementation does not exhibit input-dependent memory access patterns at cache-line granularity, i.e. 64 B. Therfore, it does not exhibit such patterns at coarser granularities, such as page granularity which is 4 KB. However, as different parts of the trusted application's code are stored on different pages during execution, monitoring page accesses can reveal the control flow of the application and therefore also sensitive data. Van Bulck *et al.* [6] have demonstrated that they are able to extract EdDSA session keys of an unmodified cryptographic library running inside an SGX enclave. Their attack is executed by continuously monitoring *Page Table Entries* while the victim application is running. There are, as explained before, certain operations that introduce control flow that depends on their input. The possible branches of these operations, however, are limited to a single operation and therefore very short. Hence, although possible, it is highly unlikely that these branches are stored on different code pages, meaning page table-based attacks will most likely not reveal any information about input data.

Branch Shadowing Attacks. The control flow leaked through code page accesses is rather coarse. Branch shadowing, however, allows an attacker, to infer fine-grained control flow information of an enclave. As Lee *et al.* [34] show, an attacker can exploit the branch prediction of modern processors to reveal sensitive data held by an enclave. Using their attack, Lee *et al.* are able to recover 66% of a private RSA key after having observed a *single* decryption. The attack is executed as follows. After analysing the victim source code, the attacker writes shadow code that aligns with the section of the victim code she wishes to attack. Finally, after the victim code has run, the attacker can use the branch history of the victim code to infer fine-grained control flow information by monitoring the branch predictions while executing her shadow code. Due to this, even short branch executions, e.g. only consisting of single operations, become visible. As a result, an attacker can use branch shadowing attacks to attack certain operations of our proof-of-concept implementation. Note, however, that this attack surface is very limited as it only reveals if a certain value is larger than another. Moreover, as the affected branches are so simple, we can adapt our implementation to hide branch accesses, as proposed by Lee *et al.* [34], in future versions.

Spectre Attacks. In 2018 Kocher *et al.* [30] developed a new attack, called Spectre, which exploits speculative execution capabilities of modern CPUs to extract secrets. This attack also translates to SGX, as reported by Chen *et al.* [9]. Using their attack, Chen *et al.* were able to extract different keys, such as sealing key and attestation key, used by SGX. By stealing the attestation key, one can forge the quotes used for remote attestation. Intel has since updated its SGX SDK to mitigate Spectre attacks against SGX [26]. Nonetheless, application developers have to analyse and potentially update their own source code to fully mitigate Spectre attacks. As we are using features of SGX that might expose Spectre vulnerabilities for our proof-of-concept implementation, more analysis and possible source code updates are necessary before we can confidentially claim to not be prone to Spectre attacks.

Timing Attacks. A naive attack against our proof-of-concept implementation is to time the execution times of different input data in order to infer properties about input and output data. For our test, we used the MNIST data set, which contains images of handwritten digits and therefore defines 10 different labels. We randomly selected one representative of each label, i.e. digit, and timed their execution times over $10,000$ repetitions. Figure 6.2 illustrates the distribution of execution times of each of these randomly selected representatives. As we can see,

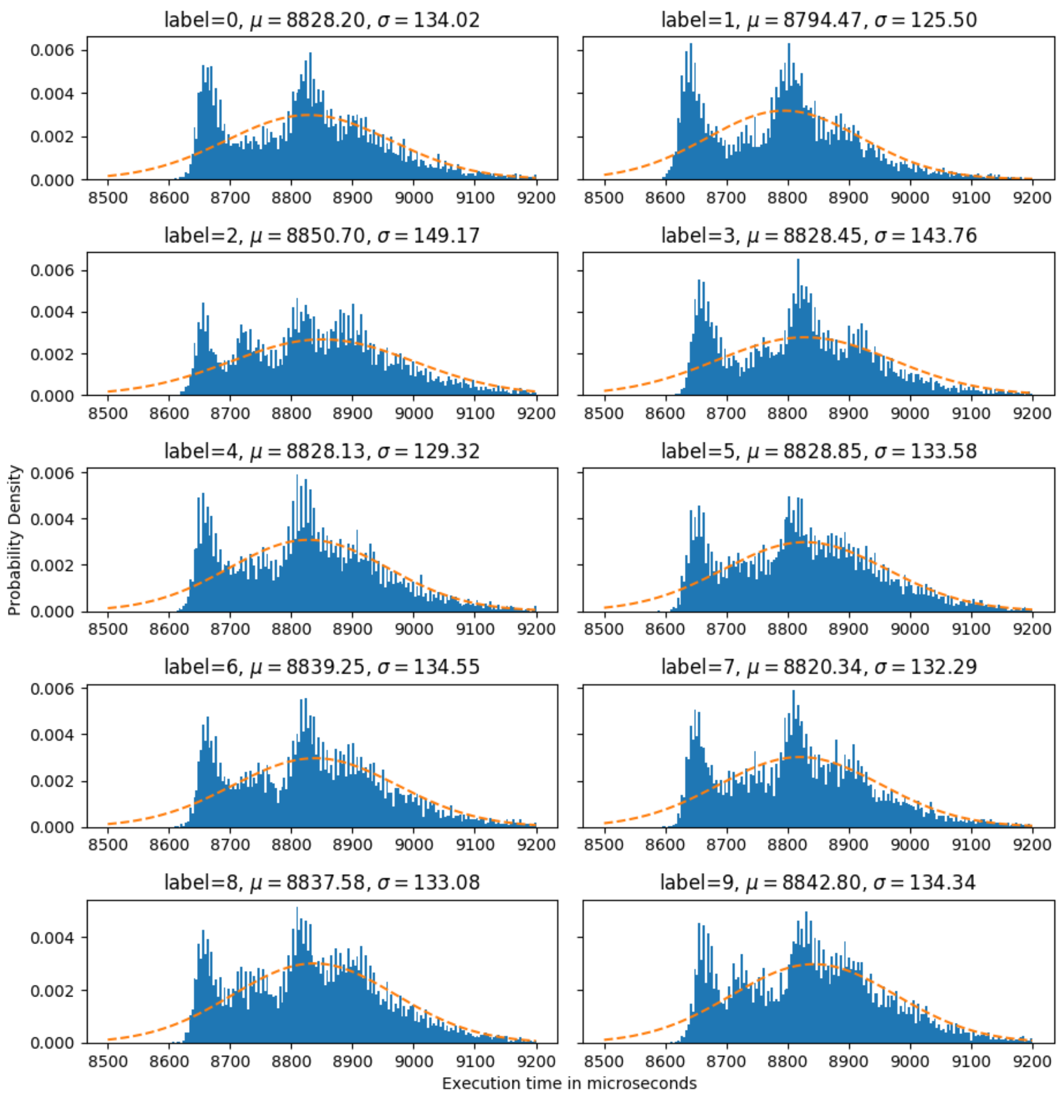

Figure 6.2. Execution times of different input images of the MNIST data set

the distributions of all tested images are close to one another. Moreover, mean μ as well as standard variation σ are very similar across all distributions. This is further illustrated by figure 6.3, which compare the "best fit" lines of the different distributions shown in figure 6.2. The lines depicted by figure 6.3 are plotted using the μ and σ of each individual distribution. Therefore, we conclude that in our test, timing execution times for different input data was not sufficient to leak information about input or output data. Although we only tested this for one model using the MNIST data set, we expect this to translate to other data sets and models as well. The reason for this is that, given a trained model $\mathcal{M}$, all calculations defined by $\mathcal{M}$ are identical for all given inputs.

Trusted Computing Base. The trusted computing base (TCB) of a system comprises the hardware and software components of said system that have to be completely trusted [3]. Therefore, we want to reduce the size of the TCB to minimise its attack surface. In the case of SGX, we can only influence the size of our

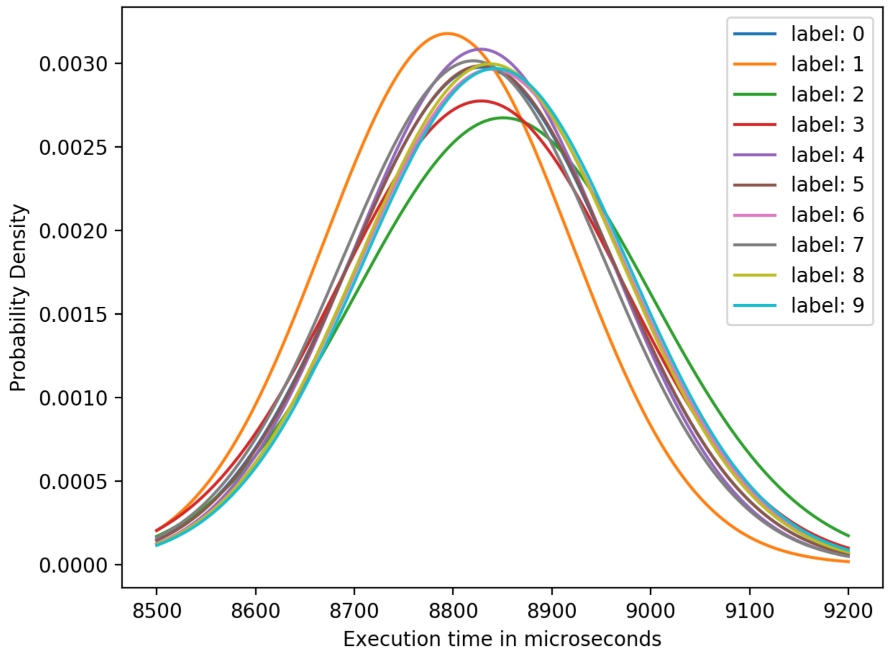

Figure 6.3. "Best fit" lines of execution distributions of different input images of the MNIST data set

TCB with the design of our enclave. As a result, the code of our enclave should be small and thoroughly evaluated to reduce the risk of software bugs which might exhibit security vulnerabilities. As mentioned in section 5.3, we created our own DNN library instead of porting an existing state-of-the-art framework or utilising an existing port of the Eigen library for SGX. Apart from the technical difficulties involved in porting a complex framework such as PyTorch or Caffe2, the decision to create a new DNN framework was also influenced by the goal of minimising the TCB of our solution. Measuring the size of existing non privacy preserving frameworks and libraries in *lines of source code* (LoC), yields the following results[1]:

- PyTorch: $\sim 1,700,000$ LoC
- Caffe2: $\sim 270,000$ LoC
- TensorFlow: $\sim 1,300,000$ LoC
- Eigen SGX port: $\sim 116,000$ LoC
- Slalom (explained in section 8.3): $\sim 130,000$ LoC
- MLCapsule (explained in section 8.3): $\sim 130,000$ LoC

In comparison, the DNN library and enclave code of our proof-of-concept implementation consists of $\sim 4,500$ LoC. The vastly smaller code base of our solution allows us to find and fix software bugs easier and faster. Moreover, a small TCB

[1]LoC numbers were acquired by measuring the source code hosted on GitHub. Benchmarks, tests, documentation, and other utility files were excluded from the measurements.

makes it easier for users and other third parties to audit as well as evaluate the source code, allowing them to determine whether it is trustworthy or not.

6.2 Functionality

The second set of requirements we defined are functional requirements which we described in section 3.3.2.

Accuracy (F-1)

Given an existing model, we wanted to assess whether its accuracy decreases if used with our proof-of-concept implementation. For our test, we trained a model in PyTorch using $60,000$ training images of the MNIST data set. The trained PyTorch model achieves a 99.14% accuracy when tested with the $10,000$ test pictures of the MNIST data set, meaning it mispredicts 86 test images. Afterwards, we exported said model to ONNX format in order to use it with our proof-of-concept implementation. Both the PyTorch code and the ONNX graph for our used model can be found in appendix B.

Using the exported model, our proof-of-concept implementation achieves the same accuracy of 99.14% with 86 mispredctions when operating on the same test set as PyTorch. In addition, consecutive runs on the same input data will always produce identical output data. Although our proof-of-concept implementation did not affect the accuracy of the tested model, this might not be the case with any model. During our tests, we measured rounding errors for certain operators, such as Gemm or Tanh, which might affect the overall model accuracy. Nonetheless, the measured rounding errors of individual values never exceeded 0.2% and therefore we do not expect a significant change in model accuracy caused by our proof-of-concept implementation.

In addition, due to the design of our system, model execution inside and outside SGX are identical. Thus, when using our proof-of-concept implementation, there will be no loss in accuracy when executing a model inside SGX compared to executing it outside SGX. We expect this to translate to other applications as well. As a result, assuming a state-of-the-art machine learning framework such as PyTorch is ported to SGX in the future, its accuracy will not by affected by SGX in any way.

Coverage (F-2)

As mentioned in chapter 5, we rely on the ONNX standard for our proof-of-concept implementation. Consequently, we aim to fully support all operators defined in the standard. As of writing this thesis (October 2018), the ONNX standard defines

115 operators with 15 of these operators being in an experimental stage [43]. For comparison, Caffe2 defines 557 operators in its catalogue [7]. This shows that the ONNX standard is still missing many operators. In fact, one experimental operator defined by ONNX is named after the tensor library ATen[2]. The purpose of this operator is to directly expose operations of the ATen library when ONNX is missing its own version of an operation [43]. For our proof-of-concept implementation we implemented a subset of the total operator set defined by the ONNX standard. Implementing the majority of the remaining operators is straightforward, yet time consuming. Some operators defined by ONNX, like the ATen operator mentioned before, might be more complicated to implement, however, as they have their own dependencies.
Our proof-of-concept implementation is based on SGX. Due to the design of SGX, one must define stack and heap size in advance so that SGX can reserve enough memory for enclave execution [25]. Whenever the defined values are exceeded, execution is aborted due to memory shortage. Choosing large values for stack and heap sizes, can affect performance due to paging. SGX introduces paging if the memory allocated by the EPC exceeds 90 MB [27]. In addition, large heap and stack sizes will affect the time required to initialise and destroy an enclave as SGX has to reserve and clear more memory [32]. As a result, when executing complex machine learning models, the memory usage of our proof-of-concept implementation might exceed the EPC limit resulting in decreased performance. To mitigate this, we export model parameters and store them in untrusted memory before passing a model to the enclave. The enclave only stores the graph of a model consisting of the different operators and their attributes. During computation, the enclave reads the model parameters from untrusted memory without copying them to trusted memory, allowing us to reduce memory usage. This, however, does not address the issues of intermediary results requiring too much memory. As we discussed in section 4.2, all intermediary results must be stored in trusted memory to ensure no input or output data leaks outside the enclave. Whilst this does not pose a limitation as such on model design, it should be taken into account when designing a new model. For instance, if performance is crucial, one could trade performance for accuracy by simplifying the model so that the required memory falls below the EPC limit.

Non-Invasiveness (F-3)

The goal of ONNX is to become a new industry standard that is used to translate machine learning models from one format to another. As we already mentioned in section 2.1.4, many popular machine learning frameworks already support

[2]https://github.com/zdevito/ATen/tree/master/aten

exporting models to ONNX format. As a result, there is a high probability that an existing model can easily be converted to ONNX format. Nonetheless, the limitations above still apply. Therefore, some operators might not yet be defined by ONNX and executing complex models might require too much memory, resulting in decreased performance due to paging introduced by SGX.

6.3 Performance

To assess the performance of our proof-of-concept implementation, as we described in section 3.4.1, we compare it to the state-of-the-art machine learning framework PyTorch. For our comparison, we used the MNIST data set to train a model in PyTorch and export it to ONNX format. Afterwards, we used the $10,000$ test images to analyse performance differences between our proof-of-concept implementation and PyTorch. For our analysis, we executed PyTorch in single-threaded CPU mode. Although PyTorch allows multithreading and even provides GPU support, we decided to limit it to a single thread in CPU mode as it allows for a better comparison. This is due to the fact that our proof-of-concept implementation is also executed on the CPU in a single thread. In addition, we only measure pure execution time, meaning no pre-processing or post-processing steps, such as initialising SGX or loading data, are included in the measurement. We exclude pre- and post-processing steps because they are usually only necessary once and therefore do not reflect the performance of either system. Finally, we compare our results to the existing privacy preserving solutions MiniONN and CryptoNets.

Throughput (P-1)

To compare the throughput of our proof-of-concept implementation with PyTorch, we measured the average time needed to classify $10,000$ test images over 10 repetitions. The average execution times are depicted by figure 6.4. As we can see, using encrypted inputs and outputs inside an SGX enclave adds no overhead compared to plaintext input and output data. In addition, SGX increased average execution time by roughly 10 seconds in our test. Whilst our proof-of-concept implementation required, regardless of setting, always more than 70 seconds, PyTorch is able to finish the task in 3.74 seconds on average. Therefore, our proof-of-concept implementation, when operating on encrypted inputs, is ~ 23.3 slower than PyTorch. Using the averages shown in figure 6.4, we can calculate the following throughputs:

Plaintext, untrusted memory: ~ 129 images per second
Plaintext, SGX enclave: ~ 115 images per second

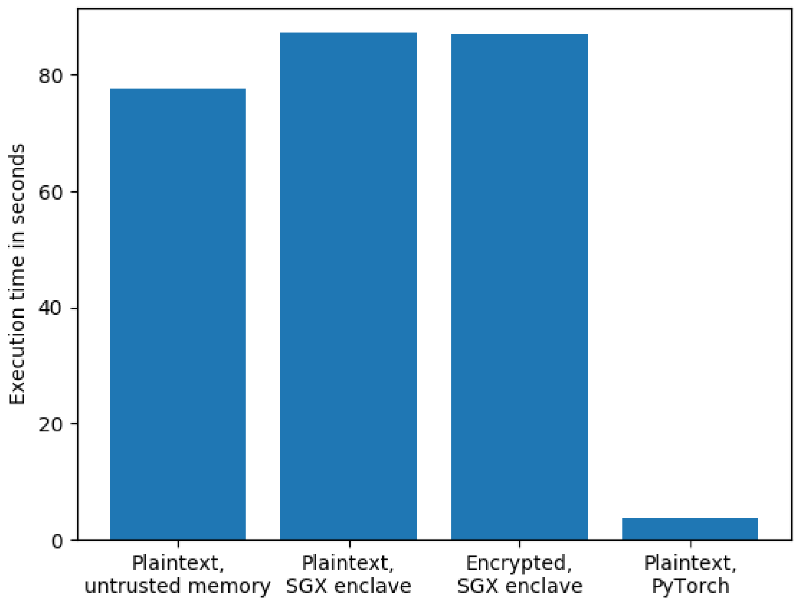

Figure 6.4. Average time required to classify all $10,000$ MNIST test images

Encrypted, SGX enclave: ~ 115 images per second
Plaintext, PyTorch: ~ 2675 images per second

Interestingly, as shown in figure 6.5, there is an initial spike in execution time when executing in untrusted memory, including PyTorch. This spike does not exist when calculating predictions inside an SGX enclave. This is probably due to the fact that SGX already copies data to CPU registers, whereas CPU caching is only introduced in subsequent runs when executing in untrusted memory.
Moreover, our proof-of-concept implementation outperforms both CryptoNets and MiniONN. CryptoNets, which is optimised for throughput, achieves $58,982$ predictions per hour, whilst MiniONN reports a throughput of $9,000$ predictions per hour [13][35]. Using the throughput shown above, our proof-of-concept implementation can calculate $414,000$ predictions per hour when operating in the encrypted setting inside an SGX enclave.

Latency (P-2)

For our latency comparison we measured the average time necessary to classify a single test image over 100 repetitions. To ensure that caching does not affect our measurement, we choose a random picture out of the MNIST test set for each iteration. The average latencies for classifying a single input image are illustrated by figure 6.6. Oddly, operating on encrypted inputs inside an SGX enclave is slightly faster than operating on plaintext inputs. Although, the difference is not significant with ~ 0.5 milliseconds, it remains unclear why this is the case. Other

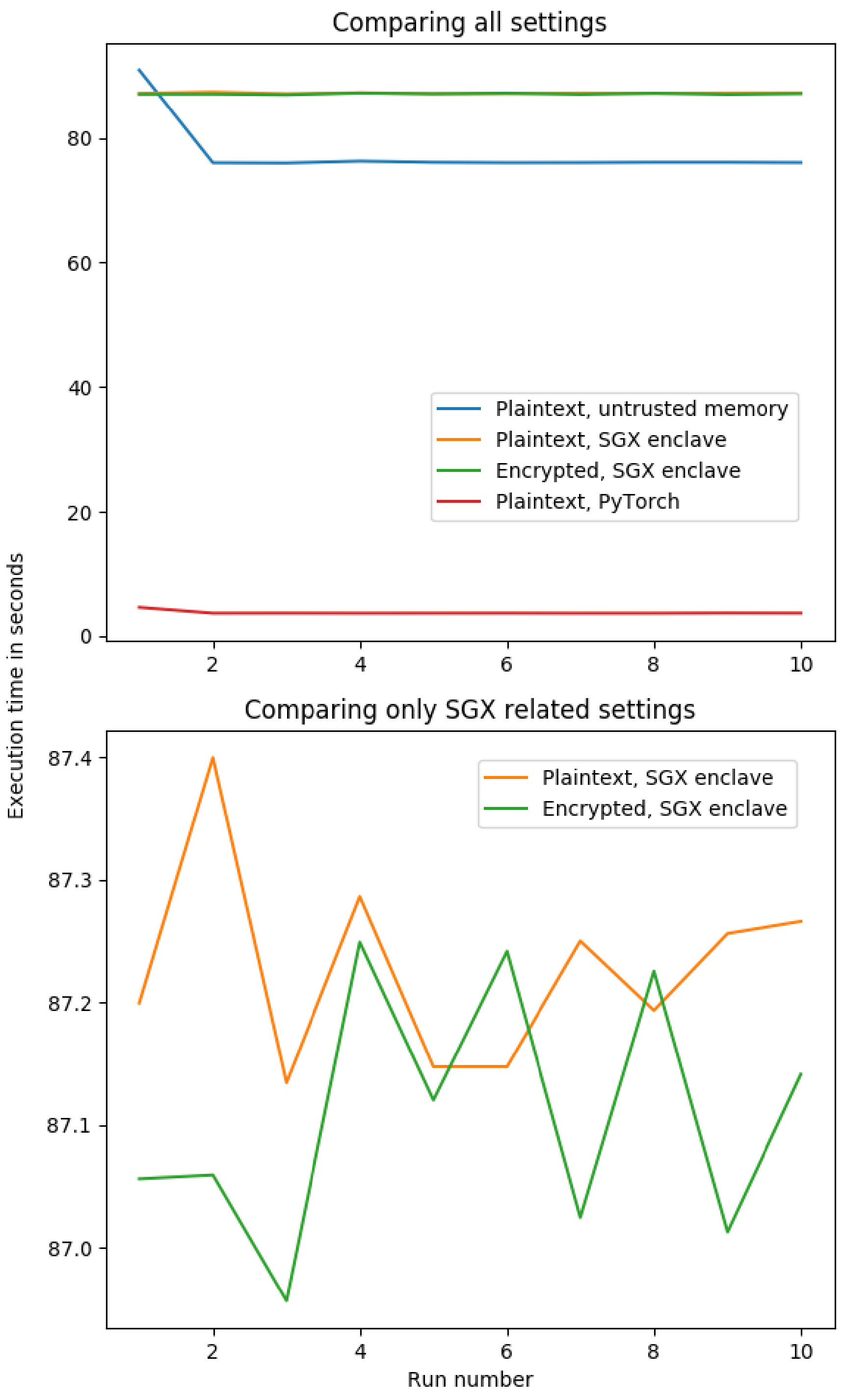

Figure 6.5. Execution times of continuously classifying all $10,000$ MNIST test images

than the additional decryption and encryption steps, the two tests are identical. In general, SGX seems to add a slight overhead of ~ 2 milliseconds to the latency. PyTorch manages to achieve a latency of ~ 0.43 milliseconds on average. Hence, when operating on encrypted inputs, our proof-of-concept implementation has a 23 time higher latency than PyTorch.

Similar to throughput, we also noticed an initial spike in latency for both non-SGX settings, as shown in figure 6.7. The reason for this temporary spike has most likely the same cause as the one seen during our throughput test.

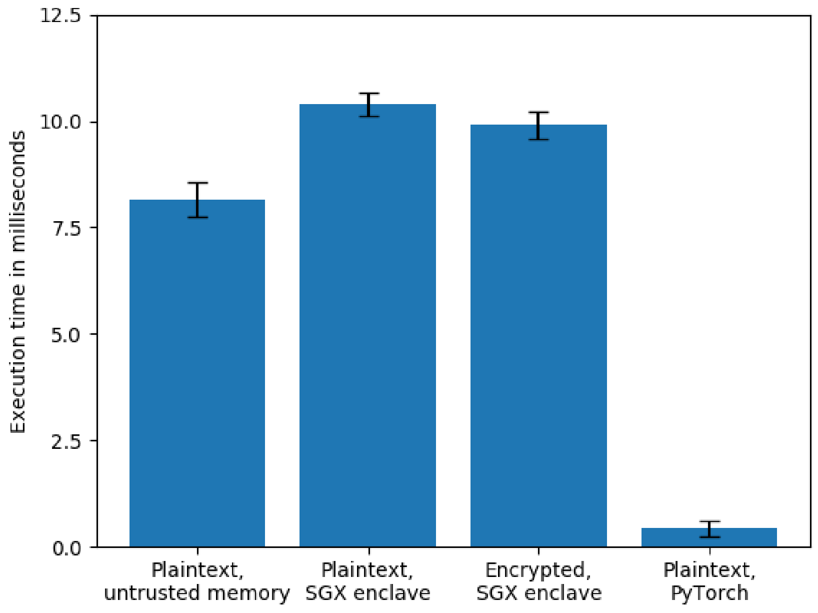

Figure 6.6. Average time required to classify a single MNIST test image

In addition, our proof-of-concept implementation also achieves lower latency than CryptoNets and MiniONN. Due to the throughput optimisation of CryptoNets, calculating between 1 and 4096 predictions at a time will always take 250 seconds [13]. MiniONN, however, manages to accomplish a much lower latency of 0.4 seconds [35]. To compute a single prediction in the encrypted setting in side an SGX enclave, our proof-of-concept implementation requires 9.9 milliseconds, i.e. 0.0099 seconds.

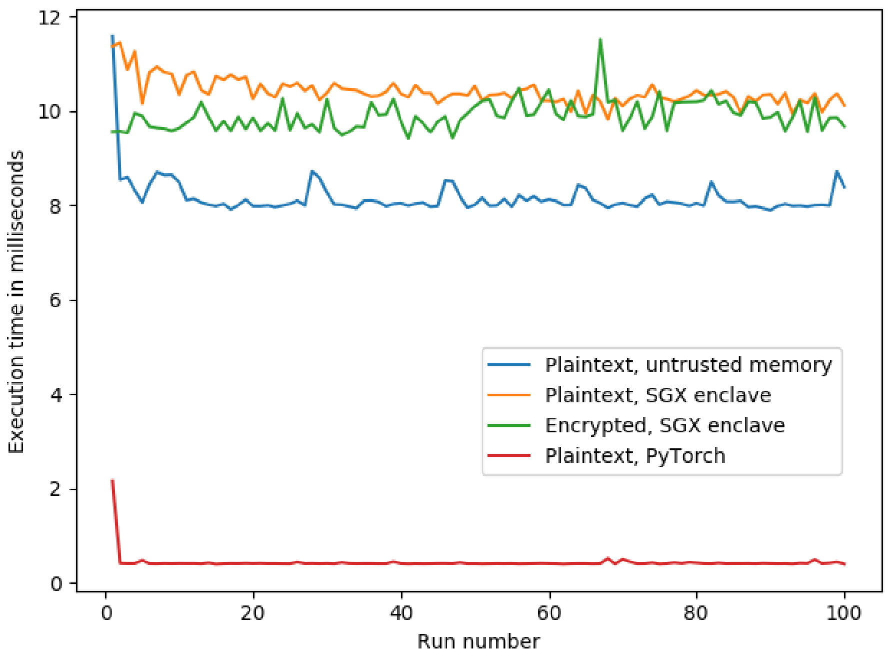

Figure 6.7. Execution times of continuously classifying a single MNIST test image

Together, figures 6.4 and 6.6 show that our proof-of-concept implementation currently is ~ 20 times slower in the same setting as PyTorch, i.e. plaintext inputs and outputs in untrusted memory, in terms of throughput and latency. Consequently, we deduce that our proof-of-concept implementation has much room for performance improvements. We assume that the overhead introduced by SGX and encryption will remain similar to what we measured in our tests, i.e. 12% overhead for throughput and 21% for latency, even after improving performance. As a result, we expect to greatly reduce the performance difference between our proof-of-concept implementation, running in SGX with encrypted inputs as well as outputs, and non privacy preserving state-of-the-art frameworks. Moreover, we anticipate ports of state-of-the-art frameworks, such as PyTorch, to have similar overheads when executed in SGX.

6.4 Deployability

We restrict our deployability evaluation to a theoretical analysis, as we only examined our proof-of-concept implementation in our own test environment. We defined our deployability goals in section 3.4.2.

Commodity Server Hardware (D-1)

Although our design does not depend on any specific TEE, we chose SGX for our proof-of-concept implementation, as explained in chapter 5. Intel SGX was first introduced in 2015 with Intel's Skylake CPUs [37]. As a result, SGX is included in all Intel CPUs since Skylake. However, software support is, at the time of writing this thesis (October 2018), limited to Windows and Ubuntu [24]. In addition, Intel also offers SGX-enabled server blocks which are intended for server applications, e.g. data centres. In fact, many large cloud providers already use server CPUs which support SGX [1][16]. Nonetheless, as of October 2018 only Microsoft Azure actively advertises its SGX capabilities, named *Azure confidential computing* [46]. Furthermore, Azure Machine Learning allows users to deploy trained machine learning models as web services [38]. Therefore, Azure could already use our proof-of-concept implementation to provide privacy preserving machine learning. Generally, a service provider is limited to using 6th generation Intel CPUs or newer when hosting his own server in order to deploy our proof-of-concept implementation. Alternatively, a service provider could also rely on existing cloud providers, such as Azure, to deploy our proof-of-concept implementation. However, if a service provider decides to use existing cloud services, he must trust his cloud provider as our proof-of-concept implementation does not protect the used model from the server.

Commodity User Hardware (D-2)

Our proof-of-concept implementation is limited to the prediction service we defined in section 4.2. In order to interact with the prediction service, the client must be able to produce the application-specific input data and possibly pre-process it. In general, most modern devices, e.g. smartphones and tablets, will be able to produce a wide array of input data, such as pictures, videos, sound and text. Additionally, pre-processing input data is generally not a very resource intensive task, hence no powerful hardware is needed. Moreover, our design only requires a TEE-enabled prediction service, meaning no specific hardware is required on the client side. Therefore, many user devices are most likely able to interact with the prediction service without the need of hardware upgrades.

7. Extensions

In this chapter we discuss possible extensions as well as improvements to our work. These concepts extend to both our design and our proof-of-concept implementation.

7.1 Security

Store Deployed Model inside the TEE

For this thesis, we assume that a service provider hosts his own model. This is not always the case, however, as large cloud providers allow third party *model owners* to deploy a model. Therefore, for this section we assume that model owner and service provider are two separate entities, where a model owner posses a trained model and a service provider deploys models as prediction services.
Our design does not protect the used model from a potentially adversarial server. Thus, a model owner is either expected to host his own server or has to trust his service provider. To address this issue, we have to protect the complete model by storing it inside the TEE. As we discussed in chapter 4 and 5, we decided to store model parameters in untrusted memory due to their potentially large memory consumption. This is necessary as most TEE implementations suffer from memory limitations. In the case of SGX, all enclaves combined can only allocate about 90 MB at any given time, after which paging is introduced which impacts performance [27]. However, this limitation is caused by the first version of the implementation of SGX. The general design of SGX can support arbitrarily large amount of memory [18]. Moreover, others, e.g. Chakrabarti *et al.* [8], have proposed strategies to improve oversubscription for secure memory. Therefore, we expect to fully protect the deployed model in future versions of our design and proof-of-concept implementation as the improvements mentioned above become more available. Re-defining the model as secret with respect to the server, however, requires substantial changes to our proof-of-concept implementation, as we have to hide memory access patterns and branch executions from the server. We expect

this to reduce performance, as hiding access patterns and branch executions introduces additional overhead [41].

Mitigate Model Extraction Attacks

In addition to protecting the model from the server, we also want to better protect it from malicious users by mitigating model extraction attacks. The existing extraction attacks discussed in section 6.1 all propose countermeasures to mitigate their appropriate attack. A promising approach proposed by Juuti *et al.* is PRADA [28]. The intuition behind PRADA is that queries submitted by a malicious user follow a characteristic distribution that differs from the distribution of benign queries. Consequently, PRADA analyses inputs over time and calculates the distance between them. If the minimum distance between inputs falls below a certain threshold, PRADA reports an attack. Therefore, PRADA can autonomously observe input data and report attacks without revealing the actual input data. As a result, we can preserve user privacy whilst monitoring user inputs to prevent model extraction attacks. As mentioned before in section 6.1, Hanzlik *et al.* [19] use PRADA for their on-device machine learning solution, allowing them to protect their deployed model from being extracted. We describe their work in detail in section 8.3. Note, that this type of input analysis is only possible because input data is available in plaintext inside the enclave. Hence, privacy preserving solutions based on encryption, such as MiniONN and CryptoNets, cannot implement this kind of input analysis to prevent model extraction attacks.
In addition, a model owner, as defined above, can utilise remote attestation to verify that a service provider implements input analysis. This gives leverage to model owners to only deploy their models after a service provider has proven that deployed models are protected from model extraction attacks by PRADA-like input analysis.

7.2 Functionality

Delete Obsolete Intermediary Results

In our evaluation, we highlighted that the memory limitations introduced by SGX pose an issue for large and complex operations which can require a significant amount of memory. Moreover, all intermediary results that are calculated as we traverse the graph are kept in memory. The reason for this is that outputs of earlier nodes in the graph will be used as inputs for later nodes. This further adds to the issue of limited memory.
To ensure memory is used more efficiently, we can delete intermediary results once

they have become obsolete. However, this is not as straightforward as deleting an intermediary result after it has been used. Depending on the graph, intermediary results might be used multiple times if the graph splits into branches. An example of a branching graph that uses intermediary results more than once is depicted by figure A.2 in appendix A. The intuition behind this is to process an input or intermediary result with different parameters. For instance, we could process an intermediary result twice using two Conv operators with filters of two different shapes. Afterwards, we can combine the outputs of both operators. In the ONNX Model Zoo[1], i.e. a collection of pre-trained, popular machine learning models in ONNX format, more than 50% for the models use branches, meaning intermediary results are processed multiple times. One possible solution to this is to add a *dry run* to the initialisation phase described in section 4.2. The purpose of this dry run is to traverse the graph once and count how many times each intermediary result is used as input. Using this information, we can introduce a counter to each intermediary result that is, similar to reference counters [48], decreased every time it is used. Once the counter reaches zero, the intermediary result is considered obsolete and deleted to free up memory.

Merge Operators

Moreover, we can further decrease the amount of memory allocated by intermediary results by combining certain simple operations with complex ones. For instance, instead of storing the output of a *reshape* operator as an extra intermediary result, we can combine it with the subsequent complex operator. This is due to how we handle `Tensor` objects internally, as discussed in section 5.3.1.

7.3 Performance

Parallel Execution

The performance results we discussed during our evaluation in section 6.3 were all achieved by single threaded execution. Although SGX supports multithreading, threads cannot be spawned within the enclave, meaning threads have to be spawned by the untrusted application and call inside the enclave [25]. As a result, adding multithreading to our proof-of-concept implementation becomes more complicated. One approach is to utilise a *master-worker pattern*. Using this pattern, the untrusted application can spawn a master thread which makes the ECALL that returns output data for given input data. However, this thread will not do any actual computations but only coordinate worker threads. After

[1]https://github.com/onnx/models

spawning the master thread, the untrusted application spawns a number of worker threads which also call inside the enclave to make themselves available for the master thread. When all worker threads are available, the master thread divides the workload among all workers and instructs them to start their computation. Once all worker threads finish their work task, the master thread combines all partial results to obtain the final result. This master-worker pattern could either be applied on the global level, meaning different workers execute different parts of the graph, or on an operator level, meaning individual operations are parallelised.

Outsource CPU Intensive Operations

Another bottleneck of our proof-of-concept implementation is its CPU limitation. Parallel processing units, such as GPUs, have been found to vastly improve performance as well as efficiency of DNN applications [11]. Consequently, state-of-the-art frameworks, such as PyTorch and Caffe2, provide GPU support to decrease execution times. As Intel SGX is limited to the CPU, our proof-of-concept limitations does not leverage GPUs and is therefore at a performance disadvantage. Tramèr *et al.* [51] have addressed this issue by outsourcing CPU intensive operations to an untrusted GPU. We describe their work in detail in section 8.3. By implementing this extension, we expect to further improve the performance results we reported in section 6.3.

8. Related Work

Privacy preserving data processing is no novel concept. Consequently, there are other existing solutions that address the same or a similar problem as we do in this thesis. For the remainder of this chapter, we discuss examples of related work that aim to provide some form of privacy preserving data processing.

8.1 Data Analysis

Data analysis allows users to run certain algorithms on their data to extract useful information. As the size of the data increases, this process is often outsourced to a remote cloud service with more computational resources where the data has to be uploaded. Therefore, we need privacy preserving data analysis solutions that allow us to perform data analysis in the cloud without revealing the uploaded data.

Opaque

Opaque, as described by Zheng *et al.* [55], is an oblivious distributed data analytics platform on top of Apache Spark[1] using SGX. Opaque is intended to be deployed in data centres to allow users to submit jobs to the cluster while preserving the users' secrets. Using remote attestation, Opaque enables users to establish a secure connections to enclaves and verify enclaves run the correct code. In addition, Opaque supports three different modes: encryption mode, oblivious mode, and oblivious pad mode. In its simplest mode, encryption mode, Opaque guarantees that all data is encrypted and authenticated. Oblivious mode, adds oblivious execution, protecting memory, disk, and network accesses of sensitive SQL operators. Oblivious pad mode, also protects input and output sizes of SQL operators as well as the chosen query plan. Zhen *et al.* report that whilst encryption mode ranges from being 52% faster to $3.3\times$ slower compared to out-of-the-box Spark SQL, oblivious modes introduce an overhead ranging from $1.6\times$

[1] https://spark.apache.org/

to $46\times$. However, they note that the performance of their system, similar to our proof-of-concept implementation, suffers from the memory limitations of SGX. As an analytics platform built on Spark, Opaque supports general data processing procedures such as SQL, graph analysis, and machine learning. As a result, Opaque is intended for users who posses data they wish to analyse as well as the source code to perform the analysis. In comparison, our solution solely focuses on machine learning and allows to deploy trained models to make them available to users who then only provide input data.

SGX-BigMatrix

BigMatrix was developed by Shaon *et al.* [49] and describes an interactive framework for performing secure data analysis in untrusted environments, such as the cloud. The framework operates on encrypted data and hides low-level operations to handle large matrices behind an abstraction layer. BigMatrix is built on SGX and uses oblivious primitives to hide access patterns. In order to support arbitrary large data sets and matrices inside an enclave, BigMatrix implements a serialisation scheme. Given an operation and data, this scheme calculates the number of blocks that are needed in memory to compute the operation as well the amount of elements than can be kept in memory. Based on this, BigMatrix partitions data or matrices into smaller blocks and serialises them individually. In general, BigMatrix comprises two components: a client application that takes inputs, such as data and tasks to perform, and a server application which interprets received input and acts accordingly. The client application includes a compiler that compiles BigMatrix's proprietary programming language to code compatible with the execution engine running on the server. In addition, BigMatrix uses remote attestation to establish shared keys, allowing clients to encrypt all input data. Similar to Chiron, BigMatrix is designed for data scientists that wish to analyse different data sets using their own source code. This contrasts our solution that solely focuses on privacy preserving predictions without the need of providing any source code or data sets.

8.2 Machine Learning using Cryptography

For this thesis, we used CryptoNets and MiniONN, which we discussed in section 2.3, as examples for existing privacy preserving machine learning solutions based on cryptography. However, there are other solutions based on cryptography that address the same problem.

SecureML

In their paper, Mohassel *et al.* [40] describe SecureML, a set of privacy preserving protocols to train neural network models, as well as other model types. Their solution allows multiple clients to share their data between two non-colluding servers, which can train different models on the joint data set of all clients without learning any information beyond the trained models. To improve efficiency, SecureML defines an offline and an online phase. All cryptographic operations are concentrated in the offline phase which uses linearly homomorphic encryption, among other cryptographic primitives, to generate shared multiplication triplets. These triplets are needed during the online phase, which consequently is void of cryptographic operations and consists entirely of integer multiplication as well as bit shifting. Apart from training neural network models, SecureML also supports privacy preserving predictions. However, this is only considered as a side product and the main focus on SecureML lies on privacy preserving training. Moreover, the latency for said privacy preserving predictions is worse than that of MiniONN, which in turn is $40\times$ slower than our solution.

GAZELLE

GAZELLE, as proposed by Juvekar *et al.* [29], describes a low latency system for privacy preserving neural network prediction. The system further improves on existing encryption-based solutions, such as MiniONN and CryptoNets, by vastly increasing performance. GAZELLE uses additively homomorphic encryption and other two-party computation primitives to ensure input privacy. In their evaluation, Juvekar *et al.* report a latency of 30 milliseconds for classifying an image using a model trained on the MNIST data set. Thus, GAZELLE is $\sim 3\times$ slower than our solution when computing predictions for MNIST. Juvekar *et al.* achieve this performance by optimising their computations using SIMD (single data multiple data) instructions for additions, multiplications, and ciphertext permutations. Nonetheless, GAZELLE suffers from similar drawbacks as other encryption-based solutions such as the inability to perform input analysis. Moreover, as explained in section 6.3, we expect our solution to have much room for performance improvements, as our proof-of-concept implementation is not optimised.

8.3 Machine Learning using TEEs

As we have shown in chapter 6, TEEs provide certain advantages over cryptography, such as allowing plaintext computations, in the setting of privacy preserving machine learning. As a result, other solutions exist that leverage TEEs to tackle different problems related to privacy preserving machine learning.

Oblivious Multi-Party Machine Learning

Ohrimenko *et al.* [41] developed a privacy preserving system for multi-party machine learning. Using their design, different parties can collaboratively train a model on an SGX-enabled data centre without disclosing their training data to any other party. Although all parties distrust one another, they can each review the source code responsible for training the model and verify that said source code is executed inside an SGX enclave. After the training code has been uploaded by one party, each party uploads their encrypted training data, followed by the encryption key once the remote attestation was successful. Using the aggregated data set, the previously uploaded code can execute and train a shared model. After the training phase is finished, each party can download the encrypted shared model. As opposed to our design, Ohrimenko *et al.* consider the SGX-enabled data centre as potentially adversarial from the point of view of the enclave. As a result, they redesigned multiple machine learning algorithms to be oblivious in order to mitigate SGX side channel attacks. Although their proposed data oblivious algorithms prevent the trained model to leak outside the SGX enclave, they potentially introduce large overheads. Whilst oblivious convolutional neural networks only add a 3% overhead, other oblivious algorithms, such as matrix factorisation, are up to $115\times$ slower than their non-oblivious counterpart. In their work, Ohrimenko *et al.* focus on secure multi-party machine learning using an aggregated data set, whereas our solution targets cloud-assisted machine learning where a user interacts with a remote model.

Chiron

Hunt *et al.* [20] proposed Chiron, a system that allows data holders to train a machine learning model on a cloud service without revealing their training data. Once training is completed, the resulting model remains in the cloud service, only allowing the original data holder to query the model via a simple interface. Moreover, Chiron protects information about the resulting model architecture and training procedures from data holders. In order to establish trust, the training source code runs inside a Ryoan [21] sandbox. Ryoan provides a distributed sandbox that is built on top of Intel SGX. As a result, Chiron can attest that training code is executed inside a Ryoan sandbox, whose code is public and which prevents the training data from being leaked, without disclosing the training code itself. To increase performance, Chiron supports launching multiple enclaves that each operate on a shard of the training data. Although all enclaves operate on different shards of training data, Chiron keeps enclaves synchronised via a parameter server to ensure all enclaves collaboratively converge to the same model.

In contrast to our solution, Chiron only allows the original data owner to query a trained model. As a result, Chiron does not allow a data owner to serve as a service provider, enabling him to charge other users to use his trained model.

Myelin

With Myelin, Hynes *et al.* [22] proposed a privacy preserving deep learning framework. Myelin is built on top of SGX and supports training models as well as computing predictions using said models. To train a model privately, a data consumer uses Myelin to compile a given model architecture into an efficient, minimised library that only includes operations needed for this particular model. Myelin does this by utilising the TVM [10] compiler. Afterwards, the data consumer deploys an SGX enclave that includes the compiled library. This enclave attests its trustworthiness to all data providers who are involved in the training process. Once the attestation phase is complete, Myelin fetches fixed-sized chunks of data from each data provider as training data. During the actual training, Myelin ensures data privacy by utilising both oblivious algorithms proposed by Ohrimenko *et al.* [41] as well as differential privacy. After the training is complete, the trained model remains private to the data consumer. To improve performance, Myelin supports multithreading inside the enclave. As a result, Myelin, using a single enclave, is able to train a model marginally faster than Chiron [20] using four enclaves on the same data set. In addition, the resulting model of Myelin has slightly better accuracy than that of Chiron, which is caused by the asynchrony between enclaves in Chiron. As for privacy preserving predictions, Myelin achieved four orders of magnitude higher performance compared to GAZELLE [29] and slightly worse performance compared to Slalom [51], which leverages the GPU. In addition, due the minimal library that is compiled for each model individually, Myelin reduces its TCB to $\sim 1,500$ LoC. It is not clear if and how Myelin would work in the setting we consider for this thesis, i.e. a service provider owns a trained model and wishes to make it available without disclosing the model or sacrificing user privacy.

DeepEnclave

Gu *et al.* propose DeepEnclave [17], a system for privacy preserving predictions for DNNs using SGX. The intuition of DeepEnclave is to partition a given DNN into two components: a *FrontNet* and a *BackNet*. Whilst the BackNet is located in untrusted memory and is not constrained by SGX, the FrontNet is located in trusted memory and therefore protected by SGX. DeepEnclave enables users to divide their DNN into FrontNet and BackNet. Afterwards, the BackNet is uploaded to an untrusted cloud provider where it is stored in plaintext. Next, when computing a prediction, the user uploads her encrypted input and the

FrontNet, in either plaintext or encrypted format, to DeepEnclave which stores them in trusted memory. After having completed a remote attestation phase, the user then uploads keys to decrypt the data. This enables DeepEnclave to compute the intermediate output of the FrontNet inside the enclave. Once this computation is finished, said intermediate output is copied to untrusted memory where the final output is computed using the BackNet. Lastly, the final output is returned to the user in plaintext. DeepEnclave's intuition to only protect part of the DNN computations is based on research by Zeiler *et al.* [54] which indicates that low layers of DNNs respond to low-level information, such as edges, whereas deep layers find more abstract information. Therefore, low layers are more tightly interconnected with the actual input data and deep layers correspond to output classes. Consequently, it should suffice to only protect FrontNet computations with SGX, as the intermediate output of the FrontNet has already been abstracted too much to reveal detailed information about the input. As a result DeepEnclave is not constrained by SGX's limited memory and can leverage more computational resources when computing the final output. This shows in the evaluation of DeepEnclave, as FrontNet computations only add a $1.6\times$ to 2.5 overhead to the baseline of computing everything in untrusted memory. In comparison to our solution, DeepEnclave expects a user to own a model and only enables offloading prediction to the cloud. Moreover, as final outputs are computed and returned in plaintext, DeepEnclave has less privacy guarantees than our system.

Slalom

Slalom was developed by Tramèr *et al.* [51] and describes a system that partially outsources DNN execution from a trusted to an untrusted but much faster processor. Slalom achieves this without compromising user privacy or data integrity. As matrix multiplications generally pose the largest bottleneck of DNN execution, Slalom delegates this operation to a fast, untrusted co-processor, e.g. a GPU. Whereas integrity of outsourced computations is achieved by utilising Freivald's algorithm, input privacy is guaranteed by adding a blinding factor to all data before sending it to the untrusted processor. As Freivald's algorithm and blinding require working over a finite field, floating point arithmetic is not possible. Thus, all inputs and model parameters have to be quantised to convert them to integers. After the computation is finished this step can easily be reversed. Moreover, Slalom defines a precomputation phase that further speeds up Freivald's algorithm. During this step, Slalom precomputes the effect of the blinding factor on the input, i.e. the "unblinding factor", as they are independent from each other. In their evaluation, Tramèr *et al.* implemented a proof-of-concept system using SGX. They found that Slalom is $6\times$ to $20\times$ faster when only providing integrity and

$3.5\times$ to $10\times$ faster when providing both privacy and integrity compared to their baseline implementation that does all computations inside SGX. Moreover, they note, that they expect even larger performance increases for TEEs that provide more memory resources than SGX.
Although Tramèr *et al.* used different data sets and model architectures than us for their performance evaluation, we expect Slalom to achieve higher performance due to its GPU utilisation. Nonetheless, Slalom uses TensorFlow for preparing models and data, meaning only models that use TensorFlow's model representation are supported, as opposed to our solution that relies on the framework-independent ONNX standard. In addtion, Slalom's main focus lies in outsourcing resource intensive computations to an untrusted party. Consequently, Slalom expects users to provide both input data and DNN when using the system. Lastly, Slalom includes an SGX-compatible port of the Eigen library which vastly increases its TCB. In total, Slalom's TCB consists of $\sim 130,000$ LoC with the port of Eigen accounting for $\sim 116,000$ LoC. The remaining source code comprises Slalom's C++ library for DNN computations. In its current form (October 2018) this library supports 7 different layer types and 3 activation functions. This is comparable to our implementation, however, out TCB is only $\sim 4,500$.

MLCapsule

MLCapsule describes a client-side privacy preserving prediction system that was published by Hanzlik *et al.* [19]. Due to its client-side deployment, MLCapsule avoids input leakage and therefore solely focuses on protecting the deployed model. MLCapsule achieves this by utilising a TEE to protect model parameters and the overall computation during prediction, meaning the user device must provide a TEE. To enable this level of protection, MLCapsule encapsulates ordinary machine learning layers in *MLCapsule layers* which contain the layer as well as the weights in encrypted form. This allows MLCapsule to execute large DNNs by loading data layer by layer into the TEE. In addition, MLCapsule includes advanced defence mechanisms to protect the deployed model from model extraction attacks. Moreover, even if the user device is offline, MLCapsule supports the pay-per-query business model and will stop computing predictions if the user runs out of funds. MLCapsule defines a setup and an inference phase. The setup phase initialises a trusted application on the user device that receives secret data, including encrypted model parameters, after having attest its trustworthiness. Afterwards starts the inference phase which requires no online connectivity. During this phase the user can query the model with her private data. When computing a prediction, MLCapsule first analyses the input to determine if its malicious or benign. If MLCapsule considers the input to be malicious, it refuses to perform any computation. Otherwise, it computes the output and possibly updates its internal state to support the pay-per-query business model. For their proof-of-concept implementation, Hanzlik *et al.* use the DNN library of Slalom that is based on Eigen and utilise SGX as their TEE. In order to mitigate the limited memory of SGX, MLCapsule encrypts large layers in small chunks of 2 MB to avoid wasting too much memory to the encrypted data that is copied to trusted memory during computation. In their evaluation, Hanzlik *et al.* used Slalom's DNN library with standard layers and plaintext weights in untrusted memory as baseline. Compared to the baseline, MLCapsule introduced an overhead of $1.02\times$ to $2.3\times$ overhead for convolutional layers and a $11\times$ to $25\times$ overhead for fully connected layers.

Hanzlik *et al.* use the same data sets and models as Slalom for their evaluation. Therefore, no direct performance comparison is possible. In contrast to our solution, MLCapsule is designed for on-device machine learning. Moreover, MLCapsule does not protect the model architecture but only its parameters. However, it implements advanced mechanisms, such as PRADA, to mitigate model extraction attacks and other DNN attacks. As MLCapsule uses the same DNN library as Slalom, we can expect that its TCB is at least $\sim 130,000$ LoC and therefore much larger than our TCB of $\sim 4,500$.

9. Conclusion

In this thesis we propose a novel system design for privacy preserving cloud-assisted machine learning by using trusted hardware. In addition, we provide a proof-of-concept implementation for our design based on Intel SGX. Moreover, we thoroughly evaluate our proof-of-concept implementation according to the goals and requirements defined in chapter 3. Our evaluation in chapter 6 shows that our proof-of-concept implementation and the TEE-based approach in general not only outperforms existing privacy preserving solutions based on cryptography, but also increases applicability by extending the functionality of said solutions. Compared to related work using TEEs for privacy preserving machine learning, we also show that our proof-of-concept implementation usses a vastly smaller TCB. Furthermore, we present possible extensions to our work. These extensions range from further improving performance as well as functionality of our proof-of-concept implementation, to extending our scope by protecting the deployed model from a potentially adversarial server. Additionally, we describe how trusted hardware enables us to leverage existing work to extend our system, for instance for input analysis, with minimum effort without compromising our security requirements. Ultimately, we further reduce the gap between privacy preserving solutions and state-of-the-art machine learning frameworks, making privacy preserving machine learning more viable for real-world applications.

Nonetheless, we also show that even by using trusted hardware, we still cannot guarantee absolute confidentiality of input, output, as well as model data without carefully adjusting our implementation. Moreover, we demonstrate that whilst trusted hardware does not impose any strict functional limitations on cloud-assisted machine learning, the limited memory provided by trusted hardware might affect the performance of machine learning algorithms. However, as these constraints are introduced by the implementation rather than the design of trusted execution environments, we expect future versions to address these issues, allowing our system and other related solutions to benefit from the improvements.

Bibliography

[1] Amazon. AWS and Intel. https://aws.amazon.com/intel/, 2018. [Online; accessed 18.07.2018].

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. 2013.

[3] N. Asokan, J. Ekberg, K. Kostiainen, A. Rajan, C. V. Rozas, A. Sadeghi, S. Schulz, and C. Wachsmann. Mobile trusted computing. *Proceedings of the IEEE*, 102(8):1189–1206, 2014.

[4] J.-P. Aumasson and L. Merino. SGX Secure Enclaves in Practice: Security and Crypto Review. *Black Hat 2016*, page 10, 2016.

[5] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017*, 2017.

[6] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium, USENIX Security 2017*, pages 1041–1056, 2017.

[7] Caffe2. Operators Catalog. https://caffe2.ai/docs/operators-catalogue.html, 2018. [Online; accessed 17.07.2018].

[8] S. Chakrabarti, R. Leslie-Hurd, M. Vij, F. McKeen, C. V. Rozas, D. Caspi, I. Alexandrovich, and I. Anati. Intel® Software Guard Extensions (Intel® SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment. In *Proceedings of the Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2017*, pages 7:1–7:8, 2017.

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR*, abs/1802.09085, 2018.

[10] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR*, abs/1802.04799, 2018.

[11] A. Coates, B. Huval, T. Wang, D. J. Wu, B. Catanzaro, and A. Y. Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013*, pages 1337–1345, 2013.

[12] C. Gentry. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM*, 53(3):97–105, 2010.

[13] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. 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*, pages 201–210, 2016.

[14] C. A. Gomez-Uribe and N. Hunt. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Trans. Management Inf. Syst.*, 6(4):13:1–13:19, 2016.

[15] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[16] Google. Intel and Google Cloud Platform. `https://cloud.google.com/intel/`, 2018. [Online; accessed 18.07.2018].

[17] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy. Securing Input Data of Deep Learning Inference Systems via Partitioned Enclave Execution. *CoRR*, abs/1807.00969, 2018.

[18] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[19] L. Hanzlik, Y. Zhang, Y. Xiao, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. *CoRR*, abs/1808.00590, 2018.

[20] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *CoRR*, abs/1803.05961, 2018.

[21] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data, 2016.

[22] N. Hynes, R. Cheng, and D. Song. Efficient deep learning on multi-source private data. *CoRR*, abs/1807.06689, 2018.

[23] Intel® Corporation. Intel SGX Details. `https://software.intel.com/en-us/sgx/details`, 2018. [Online; accessed 24.08.2018].

[24] Intel® Corporation. Intel® Software Guard Extensions. `https://software.intel.com/en-us/sgx`, 2018. [Online; accessed 18.07.2018].

[25] Intel® Corporation. Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS. `https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Reference_Linux_2.1.3_Open_Source.pdf`, 2018. [Online; accessed 25.06.2018].

[26] Intel® Corporation. Intel® Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass Side Channel Exploits. `https://software.intel.com/sites/default/files/managed/e1/ec/SGX_SDK_Developer_Guidance-CVE-2017-5753.pdf`, 2018. [Online; accessed 09.08.2018].

[27] S. Johnson, Intel. SGX - is HeapMaxSize necessary? `https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607004`, 2016. [Online; accessed 29.06.2018].

[28] M. Juuti, S. Szyller, A. Dmitrenko, S. Marchal, and N. Asokan. PRADA: Protecting against DNN Model Stealing Attacks. *CoRR*, abs/1805.02628, 2018.

[29] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium, USENIX Security 2018*, pages 1651–1669, 2018.

[30] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. *CoRR*, abs/1801.01203, 2018.

[31] K. Krawiecka, A. Kurnikov, A. Paverd, M. Mannan, and N. Asokan. Protecting Web Passwords from Rogue Servers using Trusted Execution Environments. *CoRR*, abs/1709.01261, 2017.

[32] K. A. Küçük, A. Paverd, A. C. Martin, N. Asokan, A. Simpson, and R. Ankele. Exploring the use of intel SGX for secure many-party applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016*, pages 5:1–5:6, 2016.

[33] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[34] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium, USENIX Security 2017*, pages 557–574, 2017.

[35] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pages 619–631, 2017.

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[37] R. Meghana, Intel. An overview of the 6th generation Intel® Core™ processor (code-named Skylake). `https://software.intel.com/en-us/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake`, 2016. [Online; accessed 18.07.2018].

[38] Microsoft. Deploy an Azure Machine Learning web service. `https://docs.microsoft.com/en-us/azure/machine-learning/studio/publish-a-machine-learning-web-service`, 2017. [Online; accessed 08.08.2018].

[39] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19t International Conference, Proceedings*, pages 69–90, 2017.

[40] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017*, pages 19–38, 2017.

[41] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium, USENIX Security 16*, pages 619–636, 2016.

[42] ONNX. Open Neural Network Exchange - ONNX. `https://github.com/onnx/onnx/blob/master/docs/IR.md`, 2018. [Online; accessed 27.06.2018].

[43] ONNX. Operator Schemas. https://github.com/onnx/onnx/blob/master/docs/Operators.md, 2018. [Online; accessed 17.07.2018].

[44] N. Papernot, P. D. McDaniel, and I. J. Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *CoRR*, abs/1605.07277, 2016.

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

[46] M. Russinovich. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017. [Online; accessed 18.07.2018].

[47] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, 2004.

[48] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013*, pages 93–110, 2013.

[49] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pages 1211–1228, 2017.

[50] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[51] F. Tramèr and D. Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. *CoRR*, abs/1806.03287, 2018.

[52] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing Machine Learning Models via Prediction APIs. *CoRR*, abs/1609.02943, 2016.

[53] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015*, pages 640–656, 2015.

[54] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. In *Computer Vision - ECCV 2014 - 13th European Conference*, pages 818–833, 2014.

[55] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*, pages 283–298, 2017.

Appendices

A. Graphs of Executed Models

Figures A.1 and A.2 depict the graphs of two different models we used for our evaluation. As we can see, Model A, as depicted by figure A.1, only consists of a 3 operations, whereas Model B, illustrated by figure A.2, comprises 10 nodes. Moreover, Model A accepts an input with shape $1 \times 1 \times 28 \times 28$. Therefore, Model A performs far fewer operations than Model B which defines an input shape of $1 \times 3 \times 224 \times 224$. This is why Model B requires significantly more time to predict an output, as shown in figure 6.1 in our security evaluation.
In the graphs depicted here, the top line of each node describes the operation a node performs, as well as what inputs it expects. The only exceptions to this are the very first and the very last node of each graph which simply represent the input and output tensors, respectively. All operations and their attributes are explained in more detail in section 5.3.4. Lastly, the annotations next to the arrows connecting the nodes of a graph explain how the output of the previous node is used by the next one. For instance, $Y \rightarrow A$ means that the output of the previous node, Y, is used as input A by the next node.

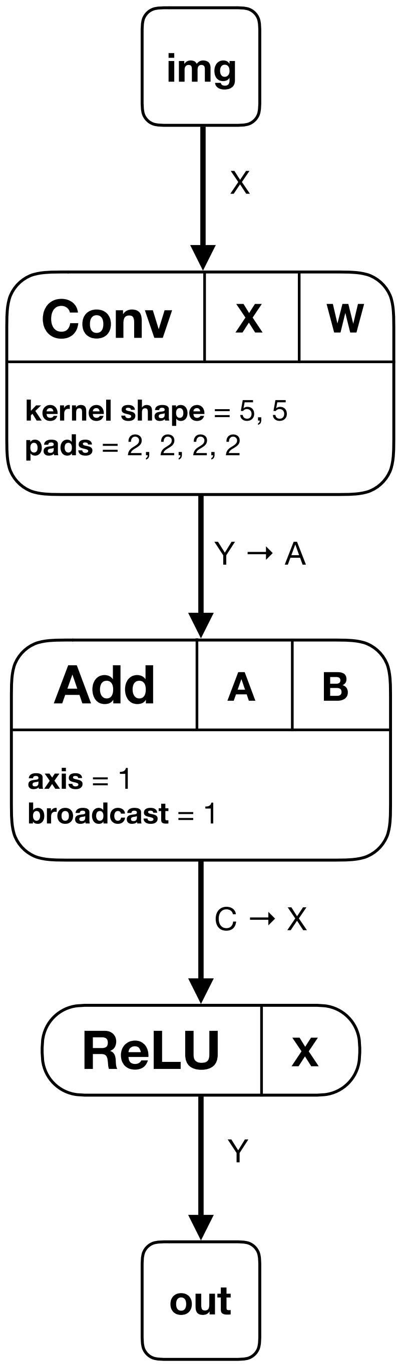

Figure A.1. Graph of Model A

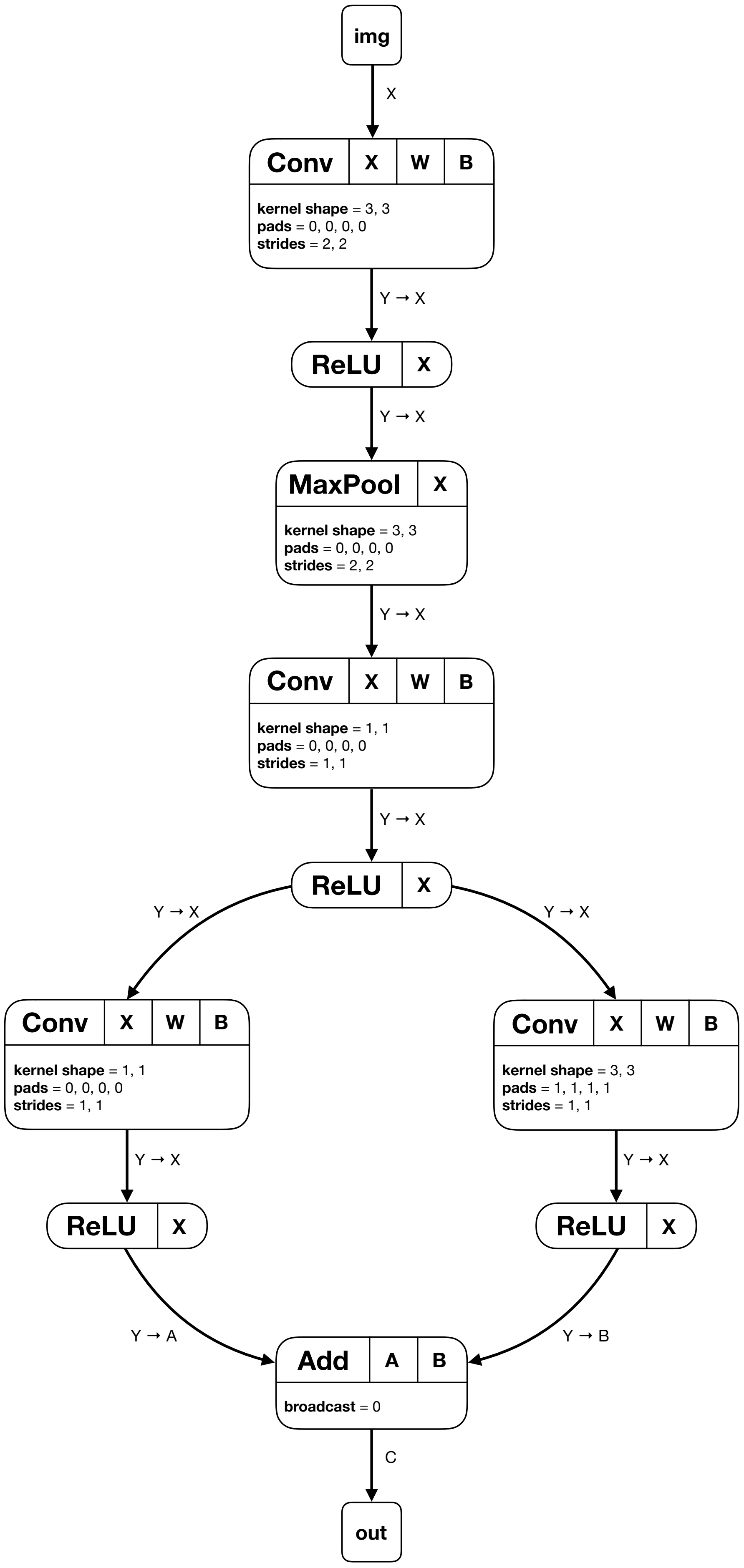

Figure A.2. Graph of Model B

B. MNIST Model

We used the code shown in listing B.1 for our PyTorch model which we trained on the MNIST data set. The resulting ONNX graph that we used for our proof-of-concept implementation is depicted by figure B.1.

```
from torch import nn

class Net3(nn.Module):
def __init__(self):
    super(Net3, self).__init__()
    self.block = nn.Sequential(
        nn.Conv2d(1, 20, 5),
        nn.MaxPool2d(2),
        nn.ReLU(),
        nn.Conv2d(20, 50, 5),
        nn.MaxPool2d(2),
        nn.ReLU())

    self.fc1 = nn.Linear(4*4*50, 500)
    self.fc2 = nn.Linear(500, 100)
    self.fc3 = nn.Linear(100, 10)
    self.relu = nn.ReLU()

def forward(self, x):
    out = self.block(x)
    out = out.view(-1, 4*4*50)
    out = self.relu(self.fc1(out))
    out = self.relu(self.fc2(out))
    out = self.fc3(out)
    return out
```

Listing B.1. PyTorch code for our MNIST model

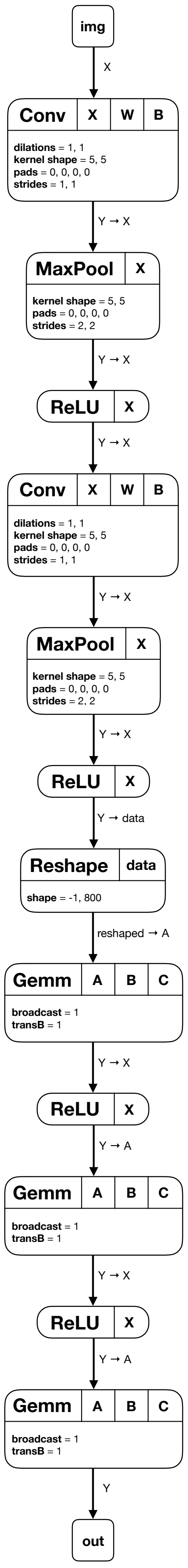

Figure B.1. ONNX graph exported from PyTorch