Attacking ML inference via malicious MPC party

Saswata Paul

Attacking ML inference via malicious MPC party

Dissertation presented to obtain the degree of Masters of Technology in Cryptology and Security
by

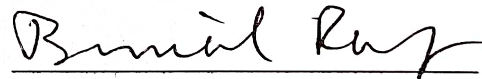
Saswata Paul
Roll - CrS2211

Promotor:
Dr. Bart Preneel, COSIC, KU Leuven
Internal Supervisor:
Dr. Bimal Kumar Roy, Indian Statistical Institute, Kolkata
Daily supervisors:
Martin Zbudila, COSIC, KU Leuven
Alireza Aghabagherloo, COSIC, KU Leuven
Dr. Aysajan Abidin, COSIC, KU Leuven

Feb-July 2024

CERTIFICATE

This is to certify that the dissertation entitled **'Attacking ML inference via malicious MPC party'** submitted by **Saswata Paul** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Cryptology and Security** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.



Bart Preneel
Professor,
COSIC
Katholieke Universiteit Leuven,
Leuven, BELGIUM.

Bimal Kumar Roy
Professor,
Cryptology and Security Research Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgments

I would like to show my highest gratitude to my supervisors, Prof. Dr. Bart Preneel and Prof. Dr. Bimal Kumar Roy for giving me the opportunity to work in COSIC, KU Leuven. I am grateful to them for their guidance and continuous support and encouragement.

I would also like to extend my gratitude to my daily supervisors, Martin Zbudila, Alireza Aghabagherloo, and Aysajan Abidin, for their invaluable suggestions and discussions. They have always motivated me with excellent insights and innovative ideas. I would also like to thank Pela Noe for her constant help during my stay in Leuven.

My deepest thanks to all the teachers of the Indian Statistical Institute, Kolkata for their valuable suggestions and discussions which added an important dimension to my research work.

Finally, I am very much thankful to my parents and family for their everlasting supports.

Last but not least, I would like to thank all of my friends for their help and support. I thank all those, whom I have missed out from the above list.

Saswata Paul
Indian Statistical Institute, Kolkata

Abstract

Secure Multi Party Computation (MPC) in a three-party honest majority setting is currently the most used cryptographic primitive for running machine learning algorithms in a privacy-preserving manner.

Although MPC typically operates with integers, it becomes necessary to extend its functionality to support machine learning algorithms, which involve arithmetic operations on decimal numbers. To address this requirement, fixed-point arithmetic is used for running machine learning algorithms. Consequently, a secure truncation protocol is essential after every multiplication to preserve precision.

Recently a maliciously secure truncation protocol named MaSTer was proposed. This protocol however lets the malicious adversary add some error with high probability to each instantiation of multiplication without getting detected.

This project aims to design an attack exploiting this vulnerability in machine learning inference from the perspective of a malicious MPC party, with a conclusion dependent on fixed-point precision. The attack method we have chosen is attacking with adversarial examples. We have given an attack strategy with a weaker assumption and discussed the results of this strategy. We have mentioned the idea of generalizing this strategy for a more general case.

Keywords: Multi Party Computation, Fixed Point Arithmetic, Truncation Protocol, Machine Learning Inference.

Contents

1 Introduction **1**
1.1 Thesis Outline 2

2 Preliminaries **3**
2.1 Cryptographic Primitives 3
2.1.1 MPC 3
2.1.2 Security setting 3
2.1.3 Secret sharing 4
2.1.4 Fixed point arithmetic and truncation 4
2.2 Machine Learning Primitives 5
2.2.1 Neural Network 5
2.2.2 Machine learning inference 6

3 MaSTer - A new truncation protocol **9**
3.1 Previous Work 9
3.2 Notation 9
3.3 Three party replicated secret sharing 10
3.4 Two party probabilistic truncation 10
3.5 Three Party Probabilistic Truncation 11
3.6 MaSTer Protocol 12
3.6.1 Protocol Description: 12
3.7 Error Analysis 14
3.8 Soundness of Consistency Check 14
3.9 Attack Idea 15

4 Towards the attack **17**
4.1 Generating adversarial images 17
4.1.1 Projected Gradient Descent: 18
4.2 Making An Inference 19
4.3 Library : fxpmath 21
4.4 The Attack 22
4.4.1 The error that the attacker can introduce 23
4.4.2 Methodology 24

4.5 Observations 26
4.5.1 Parameter 1: Success rate 27
4.5.2 Parameter 2: Euclidean distance of two final predictions . . . 31
4.6 Generalizing This Attack Strategy 34

5 Conclusion **35**

A Algorithm for Predict function **39**

List of Figures

2.1 Truncation 5
2.2 ANN [13] 6

3.1 SecureML and ABY3 protocol [19] 11
3.2 Master Protocol [20] 12
3.3 Π_{trunc} [19] 13
3.4 Consistency Check [19] 13
3.5 Attack idea 16

4.1 MNIST dataset [14] 17
4.2 Adding Perturbation [5] 18
4.3 Model Details 22
4.4 Adding ± 1 to multiplications 23
4.5 Precision vs Success_rate Plot (0-1) 27
4.6 Precision vs Success_rate Plot (2-3) 28
4.7 Precision vs Success_rate Plot (4-5) 28
4.8 Precision vs Success_rate Plot (6-7) 29
4.9 Precision vs Success_rate Plot(8-9) 29
4.10 Image classes vs Norms, Precision = 9,10 31
4.11 Image classes vs Norms, Precision = 11,12 32
4.12 Image classes vs Norms, Precision = 13,14 32
4.13 Image classes vs Norms, Precision = 15,16 33
4.14 Generalizing idea 34

Chapter 1

Introduction

Machine learning (ML) and Artificial Intelligence (AI) are widely used in various important sections of today's world. ML models require access to a high volume of data because the larger the dataset, the more accurate the output becomes. Therefore, the growth in the field of data raises concerns for data security or privacy.

Secure Multi Party Computation (MPC) is a cryptographic primitive where multiple parties jointly compute a function over their inputs keeping their inputs private. Therefore, MPC is used in the confidential training of ML models and executing ML model inference because it protects the private data of users.

On the other hand Privacy-preserving machine learning (PPML) over MPC has been a topic of research in recent years. And the three-party party honest majority is the current state-of-art in MPC over privacy-preserving ML [7].

One problem in executing ML algorithms in MPC is that MPC generally works on integers but ML models/algorithms work on decimal numbers. To address this issue, fixed point arithmetic is used, where taking some fixed precision one converts a secret-shared decimal number to an integer. Multiplying two fixed-point integers makes the length of the fractional part double, that is why we necessarily need a secure truncation protocol to keep the precision intact. SecureML [10] proposed a secure two-party probabilistic truncation allowing truncation by local operations on the shares in a semi-honest security setting. ABY3 [9] then proposed a new truncation protocol for three parties extending SecureML [10] protocol by re-sharing the result with semi-honest security. But the ABY3 protocol for malicious security setting it needed heavy pre-processing. That is why for malicious adversary a replicated secret sharing (RSS) based truncation protocol MaSTer [20] was proposed which does not require heavy pre-processing and which is secure in malicious setting for three parties.

However, this MaSTer [20] protocol lets the malicious adversary add some error without getting detected for each instantiation of multiplication.

In this thesis, we tried to design an attack exploiting this particular vulnerability having some fixed precision. We have used the MNIST dataset on a 3-layer artificial neural network to design the attack. The attack method we have chosen is attacking

with adversarial examples or perturbed images. In this attack we have tried to match the output of the MNIST dataset with the perturbed MNIST dataset by using the same vulnerability of MaSTer.

The attack strategy can be used to design a more generalized attack. The pattern of this attack can be used to design a similar kind of attack on larger networks in the future.

1.1 Thesis Outline

The topics of each chapter are described briefly here:

- In Chapter 2 we described all the preliminaries e.g., cryptographic primitives such as multi-party computation, semi-honest and malicious security setting, different kinds of secret sharing, fixed point arithmetic and truncation. Furthermore, machine learning primitives such as neural network and machine learning inference were also described.

- In Chapter 3 we discussed the new truncation protocol named MaSTer. We observed some vulnerabilities with which some attacks can be produced. We chose the attack method called "attacking with adversarial examples" and elucidated the attack idea.

- In chapter 4 we stated the neural network architecture we worked with. The generation method of adversarial examples is described thereafter. The process of making an inference, the libraries used, and the main attack methodology is described after that. We concluded the chapter by stating the observations we made from the attack.

- In Chapter 5 we stated how the attack can be generalized, and how the attacker can proceed to make the attack successful. Furthermore, we described the possible future works in this chapter.

Chapter 2

Preliminaries

In this section we first explain the cryptographic primitives such as MPC, the security setting, fixed-point arithmetic and truncation. After that we move on to the machine learning primitives such as neural network and inference.

2.1 Cryptographic Primitives

2.1.1 MPC

Multi-party computation (MPC) [18][3] or secure multi-party computation is a cryptographic primitive where the goal is to compute a function on the inputs of different parties without making the inputs public.
More formally if the parties $P_1, P_2, ..., P_n$ have inputs $x_1, x_2, ..., x_n$ then without revealing the values of x_i's ($i \in 1, 2, .., n$) parties want to compute public function $F(x_1, x_2, ..., x_n)$ by interacting with each other. The protocol of interaction should guarantee two requirements, one is **correctness** i.e; at the end of the protocol, each party gets the value of F. The other requirement is **security** which means no information about any input/data will be leaked during the protocol.

2.1.2 Security setting

Semi honest and malicious scenario:

An MPC protocol is called to be in **semi-honest security** [18] if the corrupted parties honestly participate (i.e., do not deviate from the protocol) but try to gather information from it. Achieving this type of security means the protocol is safe from information leakage.

Furthermore, an MPC protocol is called to be in **malicious security** [2] setting if the parties can arbitrarily deviate from the protocol to cheat. If a protocol is said to be secure against malicious setting, then it guarantees higher security than the semi-honest case.

Honest majority:

An MPC protocol is called to have an 'Honest majority' if more than half of the parties act honestly i.e., if there are n participating members then at least $\frac{n}{2}+1$ members honestly follow the protocol.

If there are n participants in the protocol and among them there are t participants who may not follow the protocol honestly. Honest majority occurs when $n > 2t$.

The protocol is said to have a dishonest majority if $n < 2t$,

2.1.3 Secret sharing

Secret sharing introduced by Shamir [12] refers to a method of distributing a secret among a group in such a way that no individual holds sufficient information about the secret. But when a sufficient number of parties combine their shares the secret can be reconstructed.

We will be using a particular type of secret sharing named Replicated secret sharing in this thesis.

Additive secret sharing

Additive secret sharing [17] divides a secret x into n secrets $x_1, x_2, \ldots.., x_n$ such that

$$x = \sum_{i=1}^{n} x_i \tag{2.1}$$

Replicated secret sharing

Replicated secret sharing or RSS [6] introduces a threshold to additive secret sharing by replicating shares to multiple servers i.e., one server maintains multiple shares of the secret. For example, (t,n) that is t-out-of-n RSS only needs the shares from t servers among n ones.

Several PPML algorithms try to implement RSS to improve security. Mainly 2-out-of-3 and 2-out-of-4 RSS are used in PPML schemes.

For example, in the next chapter, we will use 2 out of 3 RSS. If $s = s_1 + s_2 + s_3$, then the shares of the parties are (s_1, s_2), (s_2, s_3), (s_3, s_1) respectively.

2.1.4 Fixed point arithmetic and truncation

In recent days many MPC frameworks for privacy-preserving machine learning run on integer arithmetic in a ring $\mathbb{Z}_{2^\ell}$ e.g., [15][16] or a field $\mathbb{F}_p$[1]. One of the biggest drawbacks of using ML algorithms in MPC is it works on integers but ML algorithms work on decimal numbers. Since floating point arithmetic over MPC is expensive, fixed point arithmetic is applied.

- A real number $x \in \mathbb{R}$ is transformed into $\lfloor x \cdot 2^d \rfloor \in \mathbb{Z}$ for fixed point arithmetic, where d denotes the precision and $\lfloor . \rfloor$ denotes the floor function.
- In the case of multiplication, when two fixed point numbers with precision d are multiplied, the precision doubles. For example: $a, b \in \mathbb{R}$, $a' = a \cdot 2^d$ and $b' = b \cdot 2^d$ implies $a' \cdot b' = a \cdot b \cdot 2^{2d}$. Hence we need a **truncation** method to preserve the fixed point precision(Explained in Figure 2.1).

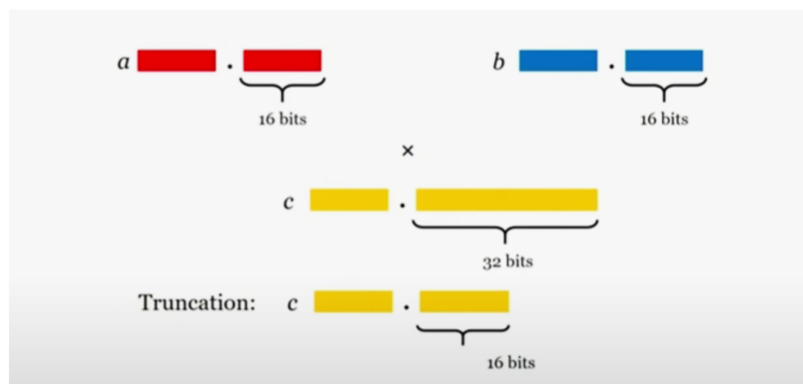


Figure 2.1: Truncation

Example: $0.6875 \cdot 1.375 = 0.9453125$
In binary: $0000.1011 \cdot 0001.0110 = 00000000.11110010$
If we take $length = 8$ and $precision = 4$. Here $length$ is the total number of bits in each number.

For truncation, we omit the first and last four logical bits (comes from the definition of truncation, and we have specified the *length* to be 8). After truncation, the result becomes 0000.1111 which becomes 0.9375 when converted to decimal.

Therefore one observation we could make is that, due to low precision truncation does not preserve the correct value of the number.

2.2 Machine Learning Primitives

2.2.1 Neural Network

Neural network (NN) [11] is a machine learning model inspired by the architecture of the human brain and is used to recognize patterns or solve complex problems that conventional machine learning models can not solve. Just like a human brain, NN has neurons that take input signals or data to give output by applying some mathematical function. A neuron or perceptron computes a weighted sum of its inputs, then applies an activation function (called 'activation' because it activates

the neuron) to this sum, and produces an output signal. Mathematically, the output y of a neuron can be represented as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.2}$$

where x_i are the input signals, w_i are the weights, b is the bias, and f is the activation function.

An artificial neural network (ANN) is a type of neural network. It has three types of

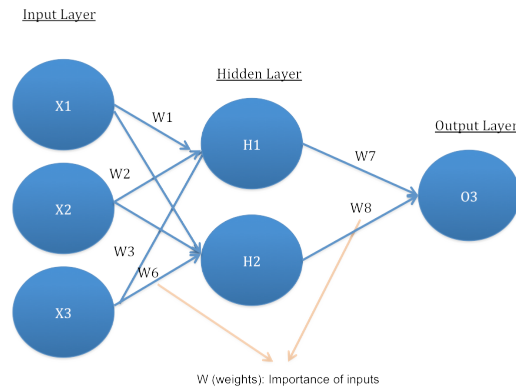


Figure 2.2: ANN [13]

layers: **input layer** that takes the inputs, **output layer** that gives the output and there can be several **hidden layers** in between input and output layers [4].

When working with ANN, two functions play an important role in the architecture: **forward propagation** and **backward propagation**. In forward propagation, the input data passes through the network layer by layer. The output of each neuron is calculated as a weighted sum of the inputs and then passed through an activation function.

Backward propagation or backpropagation is the process where the error between predicted and actual output is calculated and propagated backward through the network. The weights are then updated to minimize the error.

2.2.2 Machine learning inference

Machine learning inference is the process of running data points into some machine learning model/algorithm to get the output/prediction. Typically machine learning inference process deploys the code of machine learning algorithm into a production environment to generate predictions for inputs provided by users.

The machine learning inference process involves several steps. First, the model is trained with training data when the model learns to predict the output. Once

the model is trained it is saved to a file. In the inference phase, the saved model is loaded into the work environment, the user then inputs some data and gets the output/prediction, which means the user does not have to train the model every time. Precisely in machine learning inference, only the forward propagation function is applied. As the weights are already taken from the trained model, the backward propagation function is not needed as training is not needed.

Chapter 3

MaSTer - A new truncation protocol

In chapter 2 we have mentioned fixed point arithmetic (see sec. 2.1.4). A secure truncation protocol is needed for fixed point arithmetic to keep the precision intact. Henceforth we introduce a truncation protocol **Maliciously Secure Truncation for Replicated Secret Sharing without Pre-Processing** [20]. To comprehend this protocol it is essential to preview some previous truncation protocols, such as SecureML [10] and ABY3 [9].

3.1 Previous Work

SecureML [10] paper first introduced a truncation protocol for two parties under the assumption of semi-honest security(see sec. 2.1.2). We will be explaining this truncation protocol and the errors of this protocol in brief in Section 3.4.
ABY3 [9] paper later extended this SecureML protocol to 3 parties in the semi-honest scenario. This truncation protocol is also described in section 3.5.
However, the above-mentioned protocols are in a semi-honest security setting and the presence of a malicious adversary created problems. A new truncation protocol **Maliciously Secure Truncation for Replicated Secret Sharing without Pre-Processing** (**MaSTer**) was therefore proposed where heavy pre-processing was not required. We will discuss the MaSTer protocol in depth in this chapter and do an error analysis of this maliciously secure protocol.

3.2 Notation

We keep the notations the same as in [20]. $[\![z]\!]$ denotes 2-out-of-3 RSS, where $[\![z]\!] = (s_1, s_2, s_3)$ and $z = s_1 + s_2 + s_3$. $[\![z]\!]_i = (s_i, s_{i+1})$ for $1 \leq i \leq 3$, where $P_0 = P_3$, $P_1 = P_4$ etc. Further, $\langle z \rangle$ is used to denote 2-out-of-2 additive secret shar-

ing. Further, rshift(z, d) is used as a notation of shifting z to d logical bits right, and $\lfloor z \rceil$ is used for probabilistic truncation of SecureML [10]. $\leftarrow \$$ is used for random sampling and $r \leftarrow \$_{i,j}$ is used for uniformly random sampling of r from shared randomness between parties i and j.

Definition 1 (Fixed point encoding) *Let $-2^{\ell_x} < x < 2^{\ell_x}$ be an integer,where ℓ_x is the length of the input x. The encoding z of x in $\mathbb{Z}_{2^\ell}$ is defined as*

$$z = \begin{cases} x, & \text{if } x \geq 0, \\ 2^\ell - |x|, & \text{if } x < 0. \end{cases} \tag{3.1}$$

Definition 2 (Truncation) *Let $x \in \mathbb{Z}$ be the fixed-point encoding of x' with a fixed precision of d bits. We can write x as*

$$x = x_1 \cdot 2^d + x_2, \tag{3.2}$$

with $x_2 \in [0, 2^d)$. We define the truncation of x as

$$trc(x) = x_1. \tag{3.3}$$

3.3 Three party replicated secret sharing

3-party replicated secret sharing is used in the MaSTer protocol. Linear operations, addition and multiplication of two secret values follow from [19].

3.4 Two party probabilistic truncation

One of the main building blocks of the MaSTer protocol is the two-party semi-honest probabilistic truncation protocol proposed by SecureML [10]. This protocol mentions that if locally party P_1 computes $\langle \lfloor z \rceil \rangle_1 := \text{rshift}(\langle z \rangle_1)$ and party P_2 computes $\langle \lfloor z \rceil \rangle_2 := 2^\ell - \text{rshift}(2^\ell - \langle z \rangle_2)$, then the result is a sharing of $\lfloor x \rceil = \text{trc}(x) \pm 1$ with probability $1 - 2^{\ell_x + 1 - \ell}$. We will thoroughly discuss the errors in the next theorem.

Theorem 1 (SecureML truncation error) *Let $R \in [0, 2^\ell)$, and let z be the encoding of x (cf. Def 1). After the SecureML protocol, if we reconstruct the value z by adding the additive shares, we get,*

$$rshift(z + R, k) + 2^\ell - rshift(R, k) = trc(x) + c_1 + c_2 \cdot 2^{\ell - k} \tag{3.4}$$

where,

$$c_1 := \begin{cases} 1 & \text{if } x \geq 0 \wedge x_2 + R_2 \geq 2^k \\ -1 & \text{if } x < 0 \wedge |x_2| > R_2 \\ 0 & \text{else} \end{cases}$$

$$c_2 := \begin{cases} 1 & \textit{if } x < 0 \wedge x_1 + R_1 \geq 2^{\ell-k} \\ -1 & \textit{if } x \geq 0 \wedge |x_1| > R_1 \\ 0 & \textit{else} \end{cases}$$

Here, $x = x_1 \cdot 2^k + x_2$ *with* $0 \leq x_1 < 2^{\ell-k}$, $0 \leq x_2 < 2^k$, *and* $R = R_1 \cdot 2^k + R_2$ *with* $0 \leq R_1 < 2^{\ell-k}$, $0 \leq R_2 < 2^k$.

Furthermore, note that when $R \in [2^{\ell_x}, 2^\ell - 2^{\ell_x})$, $c_2 = 0$ *for all such* R. *The probability of a uniformly random* R *being in this range is* $1 - 2^{\ell_x+1-\ell}$. *Thus, with probability* $1 - 2^{\ell_x+1-\ell}$,

$$\textit{rshift}(z + R, k) + 2^\ell - \textit{rshift}(R, k) = \textit{trc}(x) + c_1. \tag{3.5}$$

Proof:

The proof of this theorem is mentioned in [20]. □

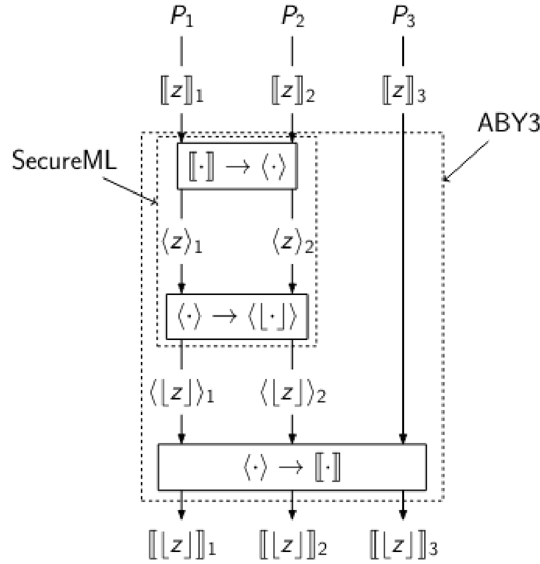


Figure 3.1: SecureML and ABY3 protocol [19]

3.5 Three Party Probabilistic Truncation

The other fundamental area of this protocol is a semi-honest 3-party probabilistic truncation protocol mentioned in ABY3 [9].
SecureML [10] truncation protocol in case of two parties have the error restricted to the last significant bit (LSB). But with a similar process if it is extended for three parties the protocol fails. Therefore ABY3 [9] proposes two new truncation protocols.

In one of them, two parties transform an RSS sharing to a 2-out-of-2 additive secret sharing, then perform SecureML with local operations and re-share the results (see from Figure 3.1). This protocol named Π_{trunc1} is described in [9].
This protocol fails in the presence of a malicious adversary. Therefore, ABY3 proposes another technique to achieve security against malicious adversaries which requires substantial preprocessing. As a result, a new truncation protocol MaSTer is proposed to address this limitation.

3.6 MaSTer Protocol

The truncation Π_{trunc1} mentioned in ABY3, when observed in a malicious setting tells that the malicious adversary can corrupt any party but only P_1 can change the share of $\lfloor z \rfloor$ when re-sharing. This is the only change possible because all other operations are local. On the other hand due to RSS other parties P_2 and P_3 already have information about z.
On a conceptual level, MaSTer [20] protocol runs ABY3 for three parties and SecureML for two parties protocol parallelly and henceforth obtains unchecked RSS shares. In the second step, the correctness/error induced by P_1 is checked by P_2 and P_3.

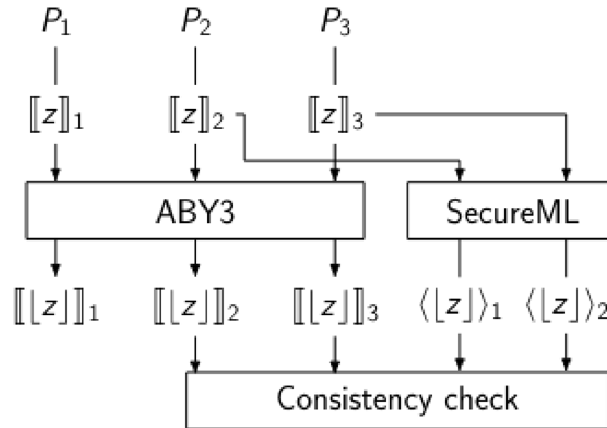


Figure 3.2: Master Protocol [20]

3.6.1 Protocol Description:

We will describe the MaSTer protocol (see Figure 3.3) next.

First round: The protocol has 2 rounds (see Figure 3.2). In the first round, there is a semi-honest truncation with re-sharing afterward. This results in *unchecked*

Truncation protocol Π_{Trunc}

P_1		P_2		P_3
$[\![z]\!]_1 = (s_1, s_2)$		$[\![z]\!]_2 = (s_2, s_3)$		$[\![z]\!]_3 = (s_3, s_1)$
$s'_1 := r' \xleftarrow{\$_{1,3}} \mathbb{Z}_{2^\ell}$				$s'_1 := r' \xleftarrow{\$_{1,3}} \mathbb{Z}_{2^\ell}$
$s'_2 := \mathsf{rshift}(s_1 + s_2) - r'$	$\xrightarrow{s'_2}$	$s'_3 := 2^\ell - \mathsf{rshift}(2^\ell - s_3)$		$s'_3 := 2^\ell - \mathsf{rshift}(2^\ell - s_3)$
		$s''_3 := r'' \xleftarrow{\$_{2,3}} \mathbb{Z}_{2^\ell}$		$s''_3 := r'' \xleftarrow{\$_{2,3}} \mathbb{Z}_{2^\ell}$
		$s''_2 := 2^\ell - \mathsf{rshift}(2^\ell - s_2)$		$s''_1 := \mathsf{rshift}(s_3 + s_1) - r''$
		$\gamma_2 = s'_2 - s''_2$	$\xrightarrow{\gamma_2}$	$\gamma_1 = s'_1 - s''_1$
		$\gamma_3 = s'_3 - s''_3$	$\xleftarrow{\gamma_1}$	$\gamma_3 = s'_3 - s''_3$
		Check $\lvert \sum \gamma_i \rvert \leq 1$		Check $\lvert \sum \gamma_i \rvert \leq 1$
Output $[\![\lfloor z \rfloor]\!]_1 := (s'_1, s'_2)$		Output $[\![\lfloor z \rfloor]\!]_2 := (s'_2, s'_3)$		Output $[\![\lfloor z \rfloor]\!]_3 := (s'_3, s'_1)$

Figure 3.3: Π_{trunc} [19]

shares of RSS shares. These *unchecked shares* go through the consistency check in the next round. The ABY3 protocol in the malicious setting lets P_1 add some error. We assume P_1 has added Δ error in the first round, P_1 sends $s'_2 + \Delta$ in the first round.

Second round: This round is called as **consistency check** (see Figure 3.4) . To detect the error added by P_1 in the first round, there is a second round, where P_2 and P_3 compute two-party truncation. Therefore P_2 and P_3 have two independently created shares of truncated values, allowing them to verify if P_1 has added some error or not.
Mathematically, P_2 and P_3 compute $\gamma_i = s'_i - s''_i$, for $i \in \{1, 2\}$ and $\gamma_3 = s'_3$ are the components of RSS shares coming from 3 party protocol and s''_i are the shares of two party truncation.

P_2		P_3
$[\![\lfloor z \rfloor]\!]_2 = (s_2, s_3)$		$[\![\lfloor z \rfloor]\!]_3 = (s_3, s_1)$
$\langle \lfloor z \rfloor \rangle_1 = s'_2$		$\langle \lfloor z \rfloor \rangle_2 = s'_1$
$s'_3 = r \xleftarrow{\$_{2,3}} \mathbb{Z}_{2^\ell}$		$s'_3 = r \xleftarrow{\$_{2,3}} \mathbb{Z}_{2^\ell}$
$\gamma_2 = s_2 - s'_2$	$\xrightarrow{\gamma_2}$	$\gamma_1 = s_1 - s'_1$
$\gamma_3 = s_3 - s'_3$	$\xleftarrow{\gamma_1}$	$\gamma_3 = s_3 - s'_3$
Check $\lvert \sum \gamma_i \rvert \leq 1$		Check $\lvert \sum \gamma_i \rvert \leq 1$

Figure 3.4: Consistency Check [19]

3.7 Error Analysis

In this section, we discuss the errors occurring in the consistency check. In the following theorem, the correctness of the consistency check is shown.

Theorem 2 (Correctness of consistency check) *Let x_1 be the result of the truncated value $x \in X$. Let $\mathbb{E}[x_1]$ denote the expected value of x_1. In an honest execution of Π_{Trunc} (see Fig. 2), the consistency check holds with probability*

$$\Pr\left(\sum_{i=1}^{3} \gamma_i \in \{0, \pm 1\}\right) \geq 1 - 2^{k+2-\ell} \cdot \mathbb{E}[x_1] \geq 1 - 2^{\ell_x+2-\ell}.$$

From this theorem, we can conclude that even with no malicious adversary the consistency check fails with certain probability mentioned above. Henceforth the choice of ℓ_x and ℓ is really important.

3.8 Soundness of Consistency Check

The soundness of the consistency check means the probability that the consistency check will detect cheating. We have assumed P_1 to be the malicious party adding error Δ.

Theorem 3 *If party P_1 adds error Δ. Then the value of Δ should satisfy the inequality $-2 \leq \Delta \leq 2$ in order to not get detected.*

Proof:

P_1 has introduced error Δ. $s_1' = trc(x) + c_1$ and $s_1'' = trc(x) + c_2$ (expressions follow from Theorem 1 and from protocol Figure 3.3). The condition for consistency check is

$$|s_1' - s_1''| \leq 1 \tag{3.6}$$

implies

$$|c_1 - c_2| \leq 1. \tag{3.7}$$

When the adversary adds an error, s_1' becomes $trc(x) + c_1 + \Delta$ and after consistency check the equation becomes

$$|c_1 - c_2 + \Delta| \leq 1 \tag{3.8}$$

From equation 3.7 and 3.8 we can come to conclusion that Δ follows the inequality

$$-2 \leq \Delta \leq 2 \tag{3.9}$$

□

This theorem indicates that the adversary can introduce an error $\Delta \in \{0, \pm 1, \pm 2\}$ without being detected with high probability. Consequently, when attempting to

cheat, the adversary has two scenarios to consider: Δ can either be ± 1 or ± 2. A malicious adversary can add $\Delta = \pm 2$ with probability $\leq \frac{1}{4}$ without getting detected (The proof of this statement is mentioned in [20]). The impact for $\Delta = \pm 2$ is larger, so the probability of detection grows exponentially.

Why 'exponentially?': Probability of the adversary adding error Δ without getting detected is $\leq \frac{1}{4}$ in each round. For one round the probability of getting detected is $1 - \frac{1}{4}$ i.e., $\frac{3}{4}$. For n rounds the probability of getting detected is $1 - (\frac{1}{4})^n$. For larger values of n the probability converges to 1.

What should be the value of Δ? In the preceding section, we highlighted a vulnerability where an adversary could introduce an error Δ during each truncation step, with $|\Delta| \leq 2$. Also, we stated that the probability of adding $\Delta = \pm 2$ error without getting caught is low. Therefore, we ignore that case.

$\Delta = 0$ means the malicious adversary is not adding any error. Henceforth, the only case that remains is $\Delta = \pm 1$. We will discuss the exact error that can be added in the next chapter.

3.9 Attack Idea

In this section, we will demonstrate the motivation for the attack and we will state what are the architectures attacker can choose to perform this attack. After that, we will discuss the overview of the strategy we took in achieving the attack.

The idea we got from the previous discussion is that the attacker can add errors to each multiplication in the architecture. Building upon this idea, an attacker can explore the potential for an adversarial attack. The attacker can choose to attack either on machine learning model training or machine learning inference. Furthermore, when selecting the model the attacker can choose any ML model like artificial neural network (ANN), convolutional neural network (CNN), residual neural network (ResNet), or any other neural network.

In this thesis, we tried to design and employ an adversarial attack on Neural Network inference and the architecture we have chosen is ANN, as it is the simplest neural network.

From the attacker's point of view: the attacker wants to tamper with the output class of the Neural Network inference. More specifically if the model inference deals with a classification problem, the attacker wants to modify the architecture in such a way that the model inference misclassifies the classes as the attacker wants. We propose a method where the adversary exploits this error injection mechanism during the multiplication operations that occur throughout the network's computations.

In an ANN, multiplication operations are fundamental as they occur during the forward propagation phase, where inputs are transformed through various layers to

produce an output. By introducing a small error Δ in each multiplication step, the adversary can subtly alter the network's internal state and change the output as needed.

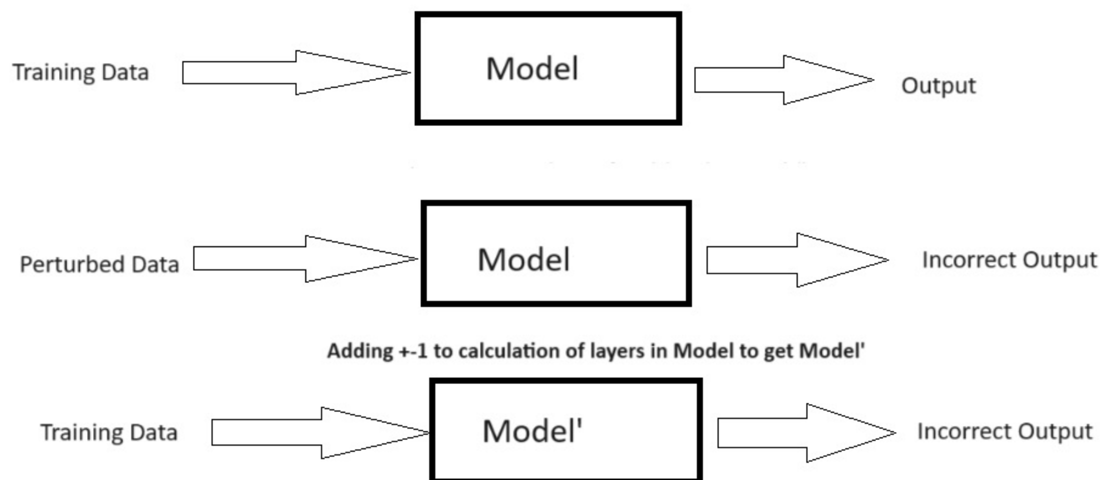


Figure 3.5: Attack idea

Idea: The attack method we have chosen is 'attacking with adversarial examples'.This technique first requires the generation of **adversarial examples** or **perturbed images**, which are slightly modified versions of normal images designed to fool the neural network and fix the architecture of an ANN. When fed normal, non-adversarial images to the ANN gives the correct output and feeding perturbed images to the ANN gives incorrect output.
Building on our earlier discussion, we understand that the adversary can modify the architecture of the ANN within certain constraints. Now we will modify the architecture to get a new model or model' (see Figure 3.5). This modified architecture aims to replicate the incorrect outputs that were originally produced by the adversarial examples but in response to the normal dataset.

Chapter 4

Towards the attack

In this chapter, we will be addressing how we approached the attack in detail and the observations we made throughout the process. Keeping the attack idea in mind (See sec 3.9) we try to implement the attack on the chosen framework such as a 3-layer artificial neural network, MNIST dataset, etc.

4.1 Generating adversarial images

The attack method we have chosen is *attacking with adversarial examples*. Therefore the first step towards the attack is generating adversarial examples. Furthermore, we have chosen the MNIST/digit-recognition dataset (see Figure 4.1).

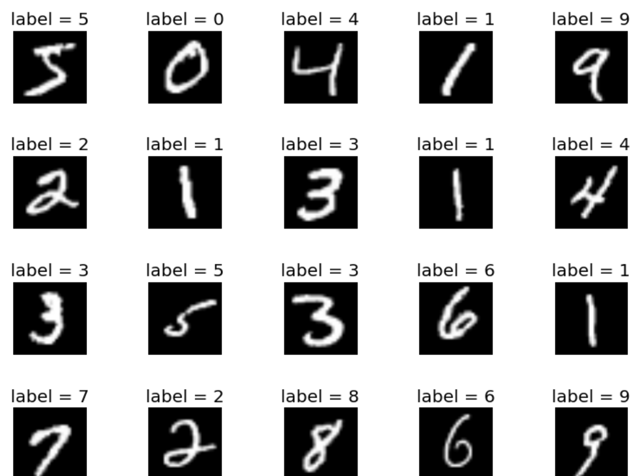


Figure 4.1: MNIST dataset [14]

We chose an attack method called *Projected Gradient Descent (PGD)*[5] (we discussed this in detail in the next section) to add perturbation to the images of the digits.

4.1.1 Projected Gradient Descent:

Projected gradient descent [5] is a method to generate perturbed images that are used to deceive machine learning models. This method iteratively applies a small amount of error/perturbation to the input images to minimize the accuracy of the machine learning or neural network models. In this attack method, we can control the amount of error we want to add to the input image.

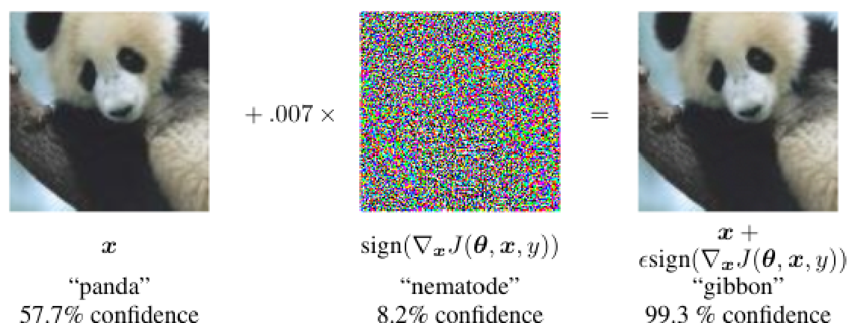


Figure 4.2: Adding Perturbation [5]

The difference between *perturbed* and *non-perturbed* images may not be visible by human eyes but the change in the prediction in neural networks is quite large (See Figure 4.2). The algorithm* of this attack is given below:

Algorithm 1 Projected Gradient Descent (PGD) Attack

1: **Input:** Original image x, true label y, model parameters θ, step size α, perturbation bound ϵ, number of iterations T
2: **Output:** Adversarial example $x^{(T)}$
3: Initialize $x^{(0)} \leftarrow x$
4: **for** $t = 0$ to $T - 1$ **do**
5: Compute gradient: $g \leftarrow \nabla_x J(\theta, x^{(t)}, y)$
6: Update adversarial example: $x^{(t+1)} \leftarrow x^{(t)} + \alpha \cdot \text{sign}(g)$
7: Project onto ϵ-ball: $x^{(t+1)} \leftarrow \text{clip}(x^{(t+1)}, x - \epsilon, x + \epsilon)$
8: **end for**
9: **Return** $x^{(T)}$

Parameters: We used the **Cleverhans** package in Python, and the parameters are taken from `https://cleverhans.readthedocs.io/en/v.2.1.0/source/attacks.html#module-cleverhans.attacks`.
Here is the list of parameters used in our code to generate perturbed MNIST images:

*This algorithm is taken from [8]

- First we specified the ann architecture(3-layer ANN).
- Input data is the test data of the MNIST dataset.
- Maximum distortion of adversarial example compared to the original input, $eps = 0.1$.
- Step size for each attack iteration $eps_iter = 0.05$.
- Number of attack iterations, $nb_iter = 10$.
- Minimum input component value, $clip_min = 0.0$
- Maximum input component value, $clip_max = 1.0$

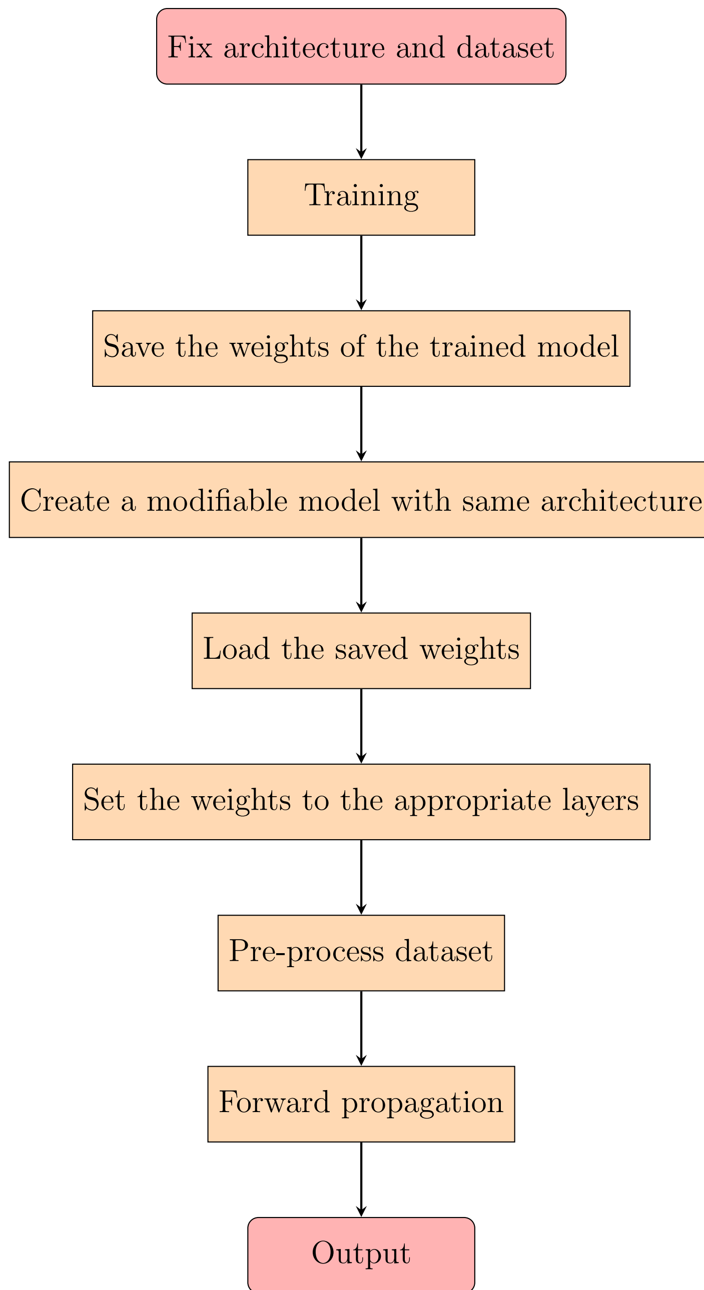
With this parameter setting, we generated the adversarial examples which will be used as the input to our machine learning inference.

4.2 Making An Inference

The next step after generating the adversarial examples is setting the environment of the attack by creating a machine learning inference.

For making the inference one crucial consideration is that the ANN model or architecture we will be using should be adjustable, i.e., we should be able to modify the architecture as we require. The workflow for making the inference is given below (see sec. 4.2).

Workflow: To create the inference we first select the ANN architecture (we fix the hidden layer's number and the activation functions) and we fix the dataset we want to work on i.e., the MNIST dataset. We train the model we have selected with the dataset of our choice. This model is written with the 'TensorFlow' library in Python. After training, we save the model and the weights and biases of each layer in some external file.



A new adaptable ANN model with the same architecture as the TensorFlow model is created thereafter but without using TensorFlow (we build it from scratch because we want to modify the architecture whenever it is required and TensorFlow does not allow us to do that). The saved weights and biases from the previous model are loaded into the new adjustable model. Thereafter the weights and biases are set to appropriate layers. The TensorFlow library, when mentioned for n layers for a model gives weights and biases for the n layers and the weights and biases for the activation layers also with '.load_weights' and '.load_biases' functions. Therefore we load the

weights and biases accordingly to the appropriate layers.

Now the forward propagation function is applied to the appropriately pre-processed data to generate the predictions. We do not use the 'back-propagation' here because the weights and the biases are already coming from a trained model, we do not need to train further.

From now on the data is just fed to this model and we get the output. We can essentially utilize this new environment as a 'blackbox'.

4.3 Library : fxpmath

Among all the open-source libraries available for fixed-point arithmetic operations the library which was found to be best suited for our work is *fxpmath*.
This is a Python library for fractional fixed-point (base 2) arithmetic and binary manipulation with Numpy compatibility†. We desired to implement the attack and get the result for a fixed precision to observe the result of the attack strategy with different precisions. To achieve this we used the mentioned library. It provides support for fixed-point operations, including addition, subtraction, multiplication, and division while preserving precision.

The pseudo-code of using the 'fxpmath' library is given below:

```
from fxpmath import Fxp
x = Fxp(-7.25, signed=True, n_word=16, n_frac=8)
```

Explanation: In the above code we wanted to convert -7.25 to fxp object with some specifications.

- `Fxp`: The class used to create the fixed-point number.
- `Number`: The first component should contain the number we want to work on.
- `signed`: The 'signed' part is set to 'True' implies signed and 'False' implies not signed. It is a way of representing fixed-point numbers.
- `n_word`: This part indicates the total number of bits used to represent the number. Both the fractional and non-fractional parts are counted in this part.
- `n_frac`: The precision required for the work is set on this n_frac part.
- `Datatype`: The objects returned by the Fxp function are 'fxp-objects'; in this case, we can say the datatype is '$fxp - s16/8$'.

†See: `https://github.com/francof2a/fxpmath`

Parameters in our setup:

- The fxpmath library supports numpy objects, therefore we directly apply the 'Fxp' function to numpy arrays.
- In our work, we first took 'n_word' to be 16, and then had experimented for 32.
- We changed the 'n_frac' value to different numbers to conclude for different precision.

4.4 The Attack

After building the inference we will now head onto the attack methodology i.e., the idea of the attack and the approach towards the attack. We have already discussed the overview of the attack in sec. 3.9.

We have mentioned that the attacker can introduce an error $\Delta = \pm 1$ in each instantiation of a multiplication. Furthermore, we have chosen the neural network architecture to be a 3-layer artificial neural network. There is a input flatten layer, after that the number of nodes of the layers are 32, 16, 10 (see Fig. 4.3).

The model that we worked on is given below:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_3 (Dense)             (None, 32)                25120

 dense_4 (Dense)             (None, 16)                528

 dense_5 (Dense)             (None, 10)                170

=================================================================
```

Figure 4.3: Model Details

The activation functions used for this model are Rectified Linear Unit (ReLu) for all the layers except the output layer. For the output layer, we have used the Softmax function.

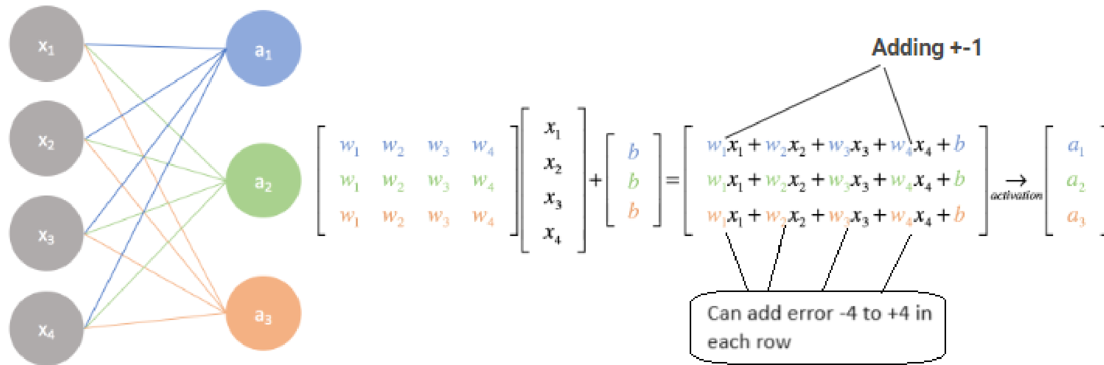


Figure 4.4: Adding ±1 to multiplications

As we have mentioned previously we are attacking the machine learning inference. Therefore, the function of ann architecture used is the forward propagation function. The attacker can introduce error to each multiplication, consequently, the attacker should focus on the multiplications in the forward propagation function (see Figure 4.4). In this figure the w's are the weights, x's are the inputs and b's are the biases.

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where $\mathbf{W} \in \mathbb{R}^{m \times n}$. That means,

$$z_j = \sum_{i=1}^{m} w_{ji} x_i + b_j \quad \text{for all } j \in \{1, 2, \ldots, n\}$$

.

When input and weight matrices are multiplied, the attacker can introduce the error with proper bound i.e., at each multiplication of w_{ji} and x_i the attacker can add ± 1. That implies the attacker can add up to $\pm n$ (where n is the number of columns of $\mathbf{W}$) in each row.

4.4.1 The error that the attacker can introduce

As we have already discussed the error bound in the paragraph above, we will now specify the bound for the architecture we have chosen.

The nodes in each layer are 32, 16 and 10 respectively. The bound on errors that can be introduced to each layer is discussed in the table 4.1.

Layer	Input size	Weight matrix size	Output size	Mult.	Error bound
1	(1,784)	(784,32)	(1,32)	784	-784 to +784
2	(1,32)	(32,16)	(1,16)	32	-32 to +32
3	(1,16)	(16,10)	(1,10)	16	-16 to +16

Table 4.1: Error bound (Mult. stands for number of multiplications in each row and error bound is for each row)

Precision factor: The goal of this thesis is to design an attack on ML inference.

As we work with fixed-point arithmetics, we have to set the precision for truncation. We first choose small precision values and then head on to larger precision values. We take care of the precision by the 'fxpmath' library (see sec. 4.3).

We assume the precision to be d. The error range $-n$ to $+n$ in floating-point representation becomes $-n \cdot 2^{-d}$ to $+n \cdot 2^{-d}$ in fixed-point representation.

Therefore we added the 'error_bound' function to keep the error in the specified range. The algorithm of this function is given below:

Algorithm 2 Error Bound

```
1:  function BOUND_ARRAY(array, bound_value)
2:      array ← NP.ARRAY(array)                ▷ Ensure the input is a numpy array
3:      for each element e in array do
4:          if e > bound_value then
5:              e ← bound_value
6:          else if e < −bound_value then
7:              e ← −bound_value
8:          end if
9:      end for
10:     return array
11: end function
```

This algorithm states how we can bind any array with a bound of our choice. We specify the bound appropriately for each layer as mentioned above.

4.4.2 Methodology

The attacker can introduce an error with a certain bound. The main idea of the attacker is to change the architecture of the model to misclassify the classes. We saw by experimenting that we can not simply add the errors to any multiplications of choice happening in the calculation of the layers to achieve our aim because it does not give the desired output. A specific choice of where to add an error is very important.

Perturbed images or adversarial examples play an important role here. Perturbed images are generated by adding perturbations to normal images, therefore the neural network gives wrong predictions for these images. We aimed to match the predictions of test image input to the modified architecture with the predictions of perturbed image input to the non-modified architecture, thereby causing misclassification of the images. To achieve this we tried to match the outputs of each layer one by one. One assumption that we had here is that we could access the output of each layer. In real life that is not the case, means the attacker does not have access of each layer. We will address this situation on the next chapter. This attack method is the first step towards generating an attack in the real-life scenario.

Overview of the attack strategy: We first sampled 100-200 images of a particular image class. Therefore, for one particular image of a digit, we have different images. Now for each of the images, we follow the below steps. If we add the error in one layer it propagates through other layers and affects the final output.

Building upon this idea we first added error to the first layer and went on to the next layers. We calculated the prediction for the perturbed image and for the test image, then determined the difference between the first layer only. We applied the 'bound_array' function (see algorithm 2) to that difference array and added it to the first layers output of prediction of the test image. This difference is the error we wanted to find. We repeated this process for the other layers. After errors in all the layers, we finally got the prediction and compared this prediction to the first prediction of the perturbed image that we had.

The algorithm of the process is described below. The 'Predict' function used in the algorithm works in such a way:

- **Input:** Two arguments, one is the data/image we want to work on and the other is the attack matrix/error matrix we want to add.
- **Output:** The output of the neural network model inference given the input.

The algorithm of 'Predict' function can be found in Appendix A.
The Python code for the attack strategy is given here: `https://github.com/Saswata2211/Attacking_ML_Inference`

Algorithm 3 Attack methodology

```
1: indices ← index of 100 images of one particular image
2: for each i in indices do
3:     attack_matrix ← [0, 0, 0, ...]                          ▷ With appropriate size
4:     Load i-th perturbed image
5:     output_perturbed, layer_perturbed ← Predict(i-th perturbed image, attack_matrix)
6:     Prediction1 ← Prediction(output_perturbed)
7:     output_test, layer_test ← Predict(i-th test image, attack_matrix)
8:     attack_matrix ← [(layer_perturbed[0] - layer_test[0]), 0, 0, ...]
9:     for j = 1, 2 do
10:        output, layer ← Predict(i-th test image, attack_matrix)
11:        attack_matrix[j] ← layer_perturbed[j] - layer[j]
12:    end for
13:    output_final, layer_final ← Predict(i-th test image, attack_matrix)
14:    Prediction2 ← Prediction(output_final)
15:    if Prediction1 == Prediction2 then
16:        return Success
17:    end if
18: end for
```

In the process, we generated the predictions and matched them. Also, we got the **attack matrices** for each of the images of each class. These attack matrices will be useful in creating the generalized attack, that works for the real-life scenario.

4.5 Observations

This section presents the key observations we got following the above-mentioned strategy. These observations are crucial in understanding the behavior of the system with respect to different precisions and different images. For a particular precision, we tried to observe some parameters for different images of an image class. We first calculated the output of perturbed images from the inference, and then applied the attack strategy to the inference architecture and fed the non-adversarial images to see the output. We selected two parameters to effectively demonstrate the performance and impact of the adversarial strategy we took. These two parameters are:

- **Success rate:** The parameter named success rate indicates the number of times the attack strategy successfully generates the same output image class for both perturbed images and non-adversarial images when the attack strategy is applied.
- **Euclidean distance of two final outputs:** This parameter is the L^2 norm or Euclidean distance between the final outputs of the output layer of perturbed

images and non-adversarial images when the attack strategy is applied.

All the plots are on $n_word = 32$ (see sec. 4.3).

4.5.1 Parameter 1: Success rate

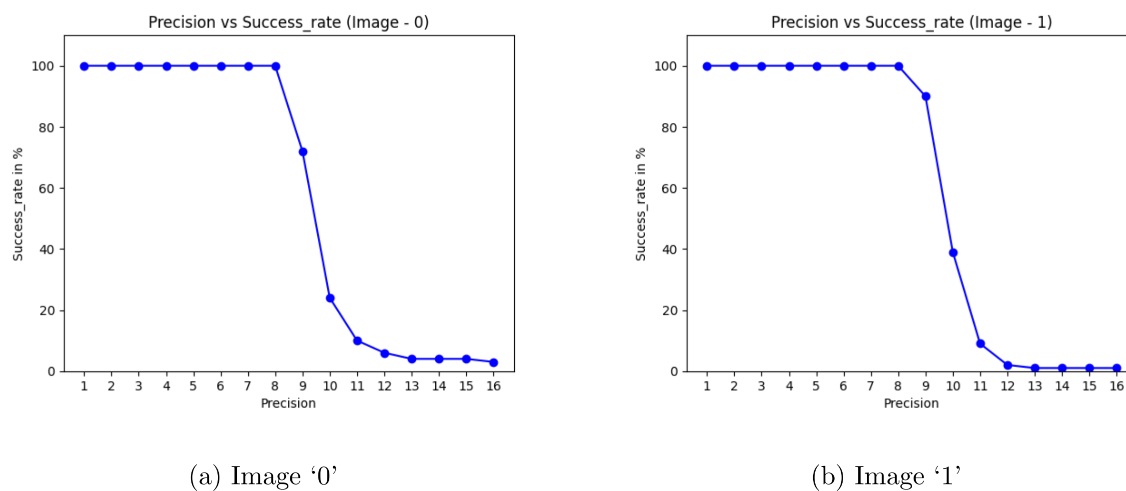
For this success rate or accuracy parameter, we calculated the success rate for each image class for a particular precision and then plotted it.

Objective

The purpose of plotting this parameter against precision is to observe how many times we can successfully predict the same output class as perturbed images and normal images after the attack.

Methodology

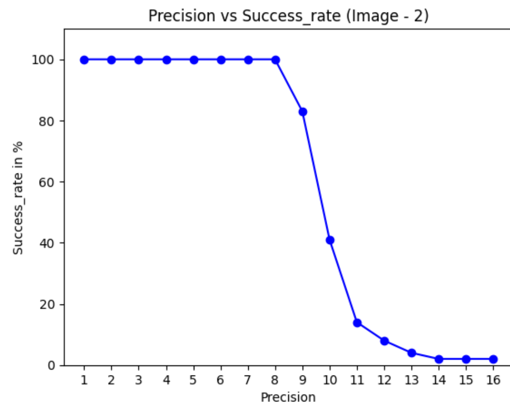
We took 200 non-adversarial images of each image class (image class '0' -'9') from the MNIST dataset. Thereafter, we checked for how many images the output classes were the same. We did this for lower precisions to higher precisions.



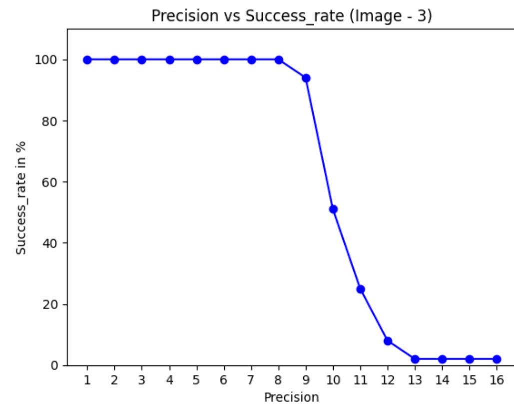
(a) Image '0'

(b) Image '1'

Figure 4.5: Precision vs Success_rate Plot (0-1)

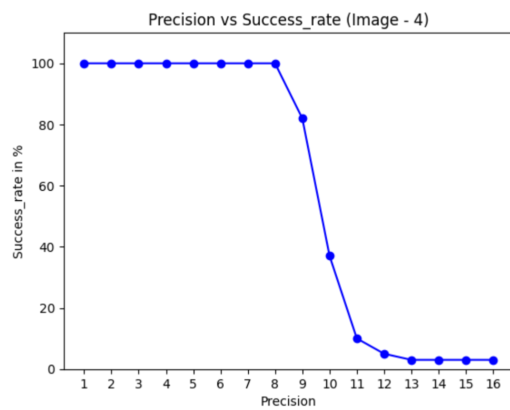


(a) Image '2'

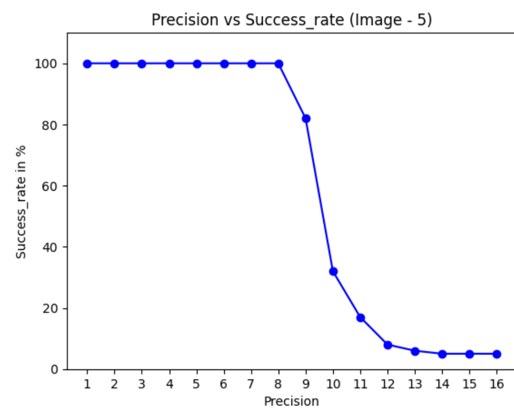


(b) Image '3'

Figure 4.6: Precision vs Success_rate Plot (2-3)

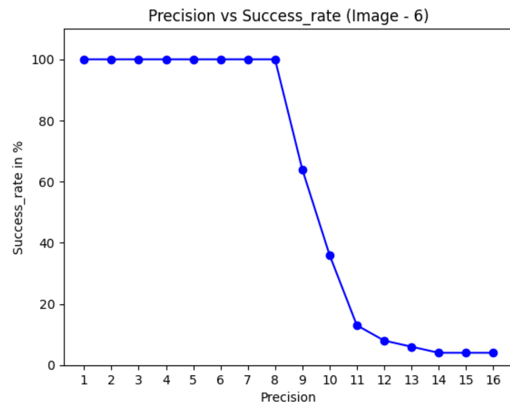


(a) Image '4'

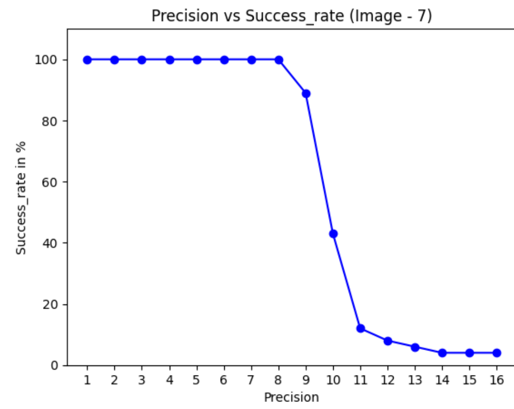


(b) Image '5'

Figure 4.7: Precision vs Success_rate Plot (4-5)

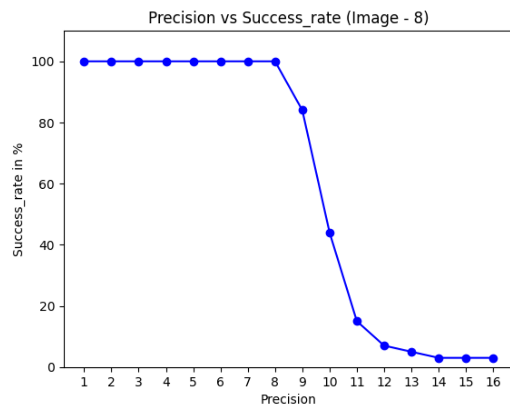


(a) Image '6'

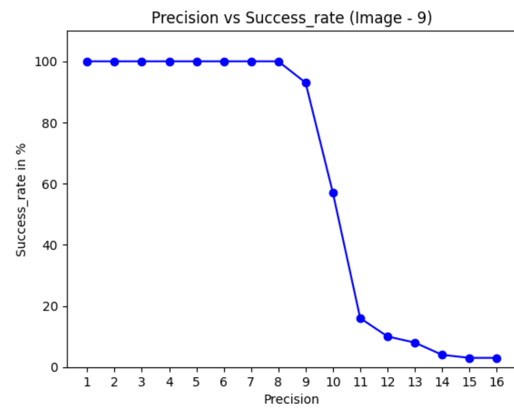


(b) Image '7'

Figure 4.8: Precision vs Success_rate Plot (6-7)



(a) Image '8'



(b) Image '9'

Figure 4.9: Precision vs Success_rate Plot(8-9)

Observations

The observations we could make from the plots are:

- For the first 8 precisions the accuracy is always 100%.
- From $precision = 9$ onwards the success rate decreases.
- After a certain precision for every image class, the success rate is close to zero and becomes constant for higher precisions than that particular precision.

Analysis

As mentioned in the attack strategy in section 4.4.2 we first add the error matrix (after bound checking) to the first layer of prediction of test data and try to match it with the first layer of the prediction of perturbed image. Then we headed on to the next layers.

Now if we could match the first layers of prediction of both images, then the next layers' outputs will also match, because the number of the nodes and activation functions are the same for both cases.

Previously we mentioned, that the error amount adversary could add for the first layer is $-784 \cdot 2^{-d}$ to $784 \cdot 2^{-d}$, where d is the precision.

One observation we could make from the attack matrices for the first layer is that,

$$|L_{Att}| < 3 < 3.0625 = 784 \cdot 2^{-8} \tag{4.1}$$

where L_{Att} is the largest element of the attack matrix. Moreover $784 \cdot 2^{-d}$ is a decreasing function for the positive values of d, so the equation 4.1 holds for every $d \leq 8$.

Therefore, as all the elements of the attack matrix of the first layer pass the bound check and stay as it is, we could fully replicate the first layer output. Therefore the accuracy is always 100%.

For the larger values of d as the bounds for error get more strict, the success rate decreases. After a particular precision (suppose 15 and above) the bounds on the bound check function become very small (as 2^{-15} is very small). Therefore naturally the success rate should become zero as we can add a very small amount of error to each layer. However, in the experiment, the success rate does not become zero but approaches zero and becomes constant at higher precision.

A potential reason for this slight error is that we use some in-built library functions in our code and those library functions do not always preserve the precision. One example that we found was in the 'ReLu' activation function, we used a numpy library function 'numpy.maximum'. This function had problems with keeping the precision intact. When we omit this function and write from scratch the success rate went closer to zero. There could be other cases like this too.

But as the number becomes constant after a certain precision, we can scale down by that constant and consider it zero.

4.5.2 Parameter 2: Euclidean distance of two final predictions

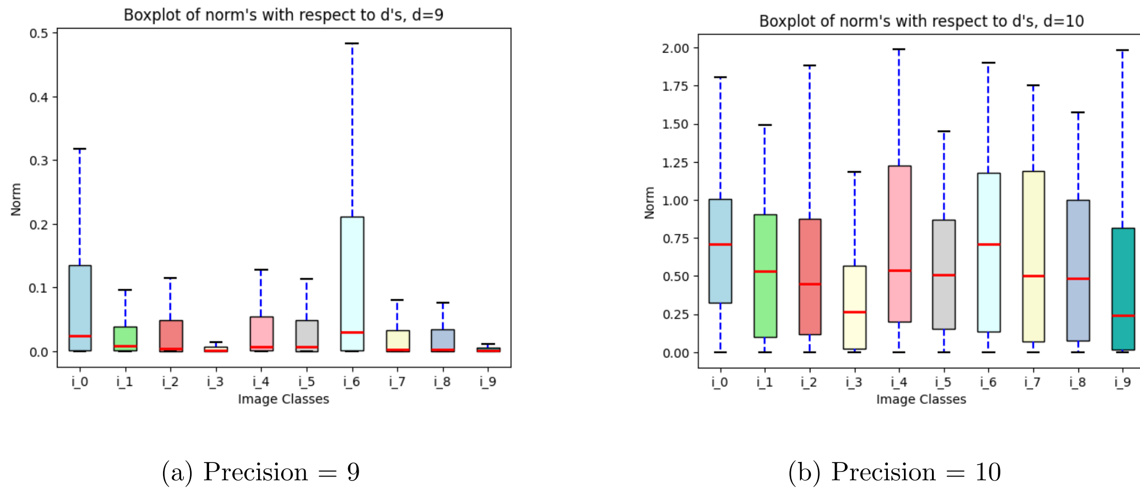
Similarly, for this parameter, we worked with different images of the same image class and with different precisions.

Objective

The objective of this plot was to measure how close we could get to matching the predictions we introduced a parameter named 'norm' or 'Euclidean distance' of these two predictions. Here we calculate the L^2 norm of the differences and plot concerning different image classes with a fixed precision.

Mehodology

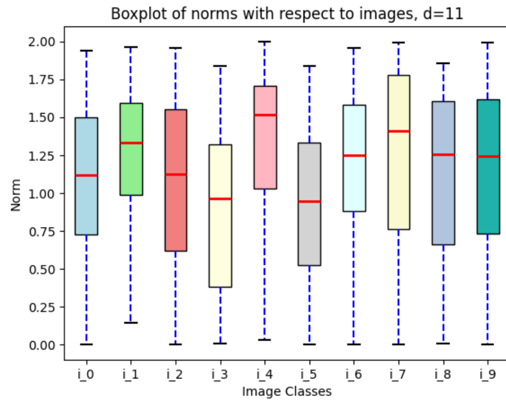
Similar to the success rate parameter we selected 200 non-adversarial images of a particular image class. For each image, we applied the attack strategy to get the final output and we got the final output of the corresponding perturbed image. Thereafter we plotted the norms in a box plot for every image for a particular precision. We plotted for $precision = 9$ onwards because for smaller precision the norms are zero because the success rate is 100% (see sec. 4.5.1).



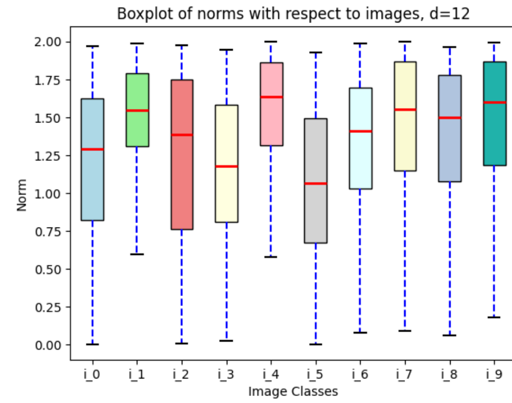
(a) Precision = 9

(b) Precision = 10

Figure 4.10: Image classes vs Norms, Precision = 9,10

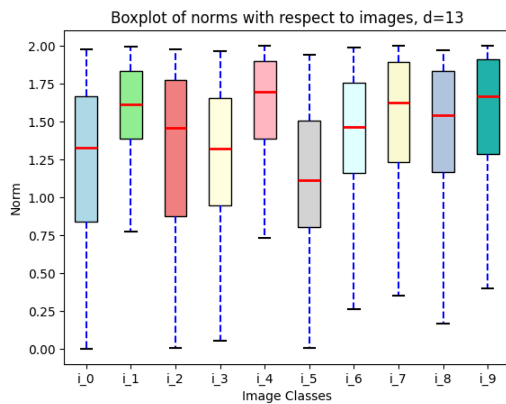


(a) Precision = 11

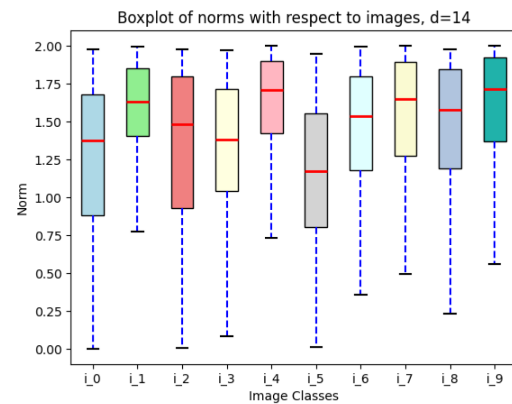


(b) Precision = 12

Figure 4.11: Image classes vs Norms, Precision = 11,12

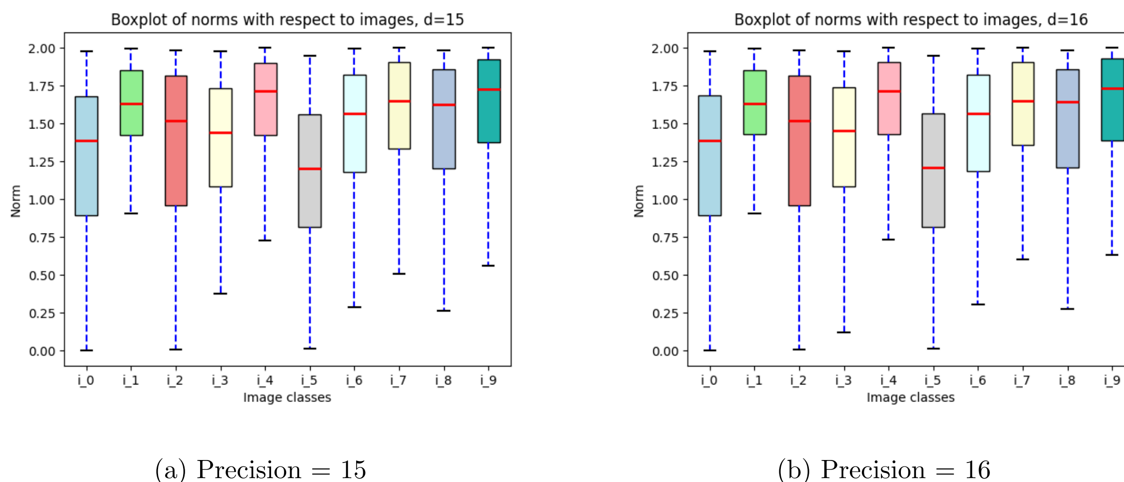


(a) Precision = 13



(b) Precision = 14

Figure 4.12: Image classes vs Norms, Precision = 13,14



(a) Precision = 15

(b) Precision = 16

Figure 4.13: Image classes vs Norms, Precision = 15,16

Observations

The observations we could make from the plots are:

- For precision = 9, the spread of norms is very small and close to zero. The median is also close to zero.
- For precision = 10 onwards the spread of norms is bigger than in the previous case. The median has also been increased.
- For a fixed precision, we could find norms in a similar range for different image classes.

Analysis: In 'Success rate vs image' plots for precision = 9 (see sec. 4.5.1), we had the observation that although the success rate for precision = 9 is not 100%, but is close to 100%. That is why the norms are closer to zero and the median is close to zero for all images.

For higher precision we observed the success rate decreases significantly, therefore, the norms also increase because the differences in the prediction of perturbed images and non-adversarial images go bigger.

Furthermore, the similar range of norms shows that the attack strategy works quite similarly for all the images.

4.6 Generalizing This Attack Strategy

Original Class	Class after perturbation	Attack matrices [For layer1], [For layer 2], [For Layer 3]
0	0	att_000 , att_001, att_002
0	1	att_010 , att_011, att_012
0	2	att_020 , att_021, att_022
⋮	⋮	⋮
1	0	att_100 , att_101, att_102
1	1	att_110 , att_111, att_112
⋮	⋮	⋮

Figure 4.14: Generalizing idea

Finally, we wanted to generalize the attack from a real-life scenario. In this previous section, we had the assumption that the attacker had access to the outputs of the layer. But in reality that is not the case. Therefore our aim was to generalize the idea/strategy mentioned previously.

In the section 4.4.2, it is mentioned that we had access to the attack matrices of each layer for each image of different image classes.

More specifically we had access to a table (see Figure 4.14) where we had the input class, the class we were able to get after the attack, and the attack matrices required for this class change. Now the attacker can sort the second column. As a result of this, the attacker can choose the class of choice from the second column and try to get some information about the attack matrices needed for this class and the attacker then can find a range of values by generalizing the attack matrices that can be added to each node of each layer of the neural network architecture. More specifically, the attacker has access to the attack matrices for each layer. The attacker can do statistical tests to find any pattern and choose the attack matrix accordingly. This way, the attacker can misclassify the image classes as the attacker wants.

Chapter 5

Conclusion

In this project, our goal was to design an attack on machine learning inference from the perspective of a malicious adversary. We focused on a protocol MaSTer, a maliciously secure truncation protocol in a three-party honest majority scenario. This protocol allows an adversary to introduce a small error of ± 1 without getting detected, in each instantiation of a multiplication.

From the attacker's perspective: the attacker aims to tamper with the output image classes of neural network inference. To demonstrate this, we selected an artificial neural network (ANN) inference process. We developed a strategy to introduce a ± 1 error into the multiplications within the ANN architecture. In this strategy, we need to align two outputs: the output from the modified ANN architecture when non-adversarial images are input and the output from the non-modified ANN architecture when adversarial images are input.

We developed the strategy assuming the attacker had access to the outputs of the ANN layers. We implemented this strategy and plotted the results using appropriate parameters. Additionally, we tried to generalize the strategy for real-life scenarios where the attacker does not have access to the outputs of the layers.

Future Work

The work presented in the thesis can open several paths for future exploration. Our current work is based on the assumption that the attacker has access to output the layers. With the help of the attack matrices produced from the attack strategy and the idea we have provided one can implement this attack in a real-life scenario.

Our current work is focused on ANN having the simplest architecture. Future work can take place for a lot more complex architecture, for example, CNN, which will help to understand the scalability of the strategy.

Furthermore, in the future one can test the proposed strategy on various state-of-the-art models, such as ResNet.

In the future, the adversarial strategy can be implemented and tested within

MPC frameworks, which will ensure the robustness of the privacy-preserving machine learning models against adversarial attacks.

Bibliography

[1] Attrapadung, N., Hamada, K., Ikarashi, D., Kikuchi, R., Matsuda, T., Mishina, I., Morita, H., Schuldt, J.C.N.: Adam in private: Secure and fast training of deep neural networks with adaptive moment estimation (2021)

[2] Canetti, R.: Security and composition of multi-party cryptographic protocols. Cryptology ePrint Archive, Paper 1998/018 (1998), `https://eprint.iacr.org/1998/018`, `https://eprint.iacr.org/1998/018`

[3] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 218–229. ACM (1987)

[4] Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge, MA, USA (2016), `http://www.deeplearningbook.org`

[5] Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples (2015)

[6] Guo, J., Shuai, M., Wang, Q., Li, W., Lin, J.: Replicated additive secret sharing with the optimized number of shares. In: Li, F., Liang, K., Lin, Z., Katsikas, S.K. (eds.) Security and Privacy in Communication Networks. pp. 371–389. Springer Nature Switzerland, Cham (2023)

[7] Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: Super-fast and robust privacy-preserving machine learning. Cryptology ePrint Archive, Paper 2020/592 (2020), `https://eprint.iacr.org/2020/592`

[8] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: International Conference on Learning Representations (2018)

[9] Mohassel, P., Rindal, P.: Aby3: A mixed protocol framework for machine learning. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. pp. 35–52 (2018)

[10] Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE symposium on security and privacy (SP). pp. 19–38. IEEE (2017)

[11] Ng, A.: Cs229 lecture notes - supervised learning (2012)

[12] Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (nov 1979), https://doi.org/10.1145/359168.359176

[13] Towards Data Science: Towards data science (2024), https://towardsdatascience.com/, accessed: 2024-06-07

[14] Toyoda, K.: Mnist dataset introduction (2016), https://corochann.com/mnist-dataset-introduction-532/, accessed: 2024-06-07

[15] Wagh, S., Tople, S., Benhamouda, F., Kushilevitz, E., Mittal, P., Rabin, T.: Falcon: Honest-majority maliciously secure framework for private deep learning (2020)

[16] Watson, J.L., Wagh, S., Popa, R.A.: Piranha: A GPU platform for secure computation. Cryptology ePrint Archive, Paper 2022/892 (2022), https://eprint.iacr.org/2022/892, https://eprint.iacr.org/2022/892

[17] Xiong, L., Zhou, W., Xia, Z., Gu, Q., Weng, J.: Efficient privacy-preserving computation based on additive secret sharing. CoRR abs/2009.05356 (2020), https://arxiv.org/abs/2009.05356

[18] Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982). pp. 160–164. IEEE (1982)

[19] Zbudila, M., Pohle, E., Abidin, A., Preneel, B.: MaSTer: (Practically) Maliciously Secure Truncation for Replicated Secret Sharing (Jul 2023), https://doi.org/10.5281/zenodo.8197660

[20] Zbudila, M., Pohle, E., Abidin, A., Preneel, B.: MaSTer: Maliciously secure truncation for replicated secret sharing without pre-processing. Cryptology ePrint Archive, Paper 2024/1026 (2024), https://eprint.iacr.org/2024/1026, https://eprint.iacr.org/2024/1026

Appendix A

Algorithm for Predict function

Algorithm 4 Predict Function

```
 1: function PREDICT(input_data, attack_matrix)
 2:     samples ← len(input_data)
 3:     results ← [ ]
 4:     layer_outputs ← [ ]
                                                ▷ Run network over all samples
 5:     for i ← 0 to samples − 1 do
 6:         output ← input_data[i]
 7:         sample_outputs ← [ ]
 8:         for layer_index, layer in enumerate(self.layers) do
 9:             output ← layer.forward_propagation(output, layer_index, attack_matrix)
10:             sample_outputs.append(output)
11:         end for
12:         results.append(output)
13:         layer_outputs.append(sample_outputs)
14:     end for
15:     return results, layer_outputs
16: end function
```

Comments

1. 'forward_propagation' is a normal forward propagation function with an attack matrix parameter added on. This matrix is the list of the errors to be added to the layers.
2. It takes the images as the input and outputs the final prediction and the outputs of each layer.